

版权注意事项：

- 1、书籍版权归作者和出版社所有
- 2、本PDF仅限用于个人获取知识，进行私底下的知识交流
- 3、PDF获得者不得在互联网上以任何目的进行传播
- 4、如觉得书籍内容很赞，请购买正版实体书，支持作者
- 5、请于下载PDF后24小时内删除本PDF。

VR 三维技术系列



三维模型 变形算法： 理论和实践（C#版本）

● 赵 辉 顾险峰 雷 娜 著



中国工信出版集团



电子工业出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY
<http://www.phei.com.cn>

VR 三维技术系列

三维模型变形算法：理论和实践

(C#版本)

赵 辉 顾险峰 雷 娜 著

电子工业出版社

Publishing House of Electronics Industry

北京·BEIJING

内 容 简 介

《三维模型变形算法：理论和实现（C#版本）》介绍了三维模型变形算法和线条抽取算法。本书分为14章，详细讲述了Blender软件中的变形、网格变形算法、外包框变形算法、均值坐标变形算法、格林变形算法、拉普拉斯变形算法、拉普拉斯矩阵在三维模型近似、光滑、优化、骨骼抽取、顶点之间最短距离算法上的应用；还有三维模型的频谱分析、骨骼动画算法、蒙皮算法、三维曲线生成算法、三维模型特征曲线的抽取算法。本书包含了三维模型处理的各种核心算法。

本书不仅可以作为数字媒体技术专业的专业基础课，还可以作为计算机学科和软件工程学科“数据结构和算法”、“计算机图形学”等课程的教材和参考书。我们提供了书里部分代码在网上的开源下载。

未经许可，不得以任何方式复制或抄袭本书之部分或全部内容。
版权所有，侵权必究。

图书在版编目(CIP)数据

三维模型变形算法：理论和实践：C#版本/赵辉，顾险峰，雷娜著. —北京：电子工业出版社，2017.7
(VR三维技术系列)

ISBN 978-7-121-31678-4

I. ①三… II. ①赵… ②顾… ③雷… III. ①三维动画软件—算法分析 IV. ①TP391.41

中国版本图书馆CIP数据核字(2017)第120538号

策划编辑：张迪 (zhangdi@phei.com.cn)

责任编辑：张迪

印 刷：中国电影出版社印刷厂

装 订：三河市良远印务有限公司

出版发行：电子工业出版社

北京市海淀区万寿路173信箱 邮编 100036

开 本：787×1092 1/16 印张：18.75 字数：480千字

版 次：2017年7月第1版

印 次：2017年7月第1次印刷

定 价：99.00元

所购买电子工业出版社图书有缺损问题，请向购买书店调换。若书店售缺，请与本社发行部联系，联系及邮购电话：(010) 88254888，88258888。

质量投诉请发邮件至 zlt@phei.com.cn，盗版侵权举报请发邮件至 dbqq@phei.com.cn。

本书咨询联系方式：(010) 88254469，zhangdi@phei.com.cn。

序

2015 年以来,虚拟现实技术的应用在国际国内发展很快。教育、医疗、娱乐、影视、游戏、安全、交通等各行各业都对虚拟现实技术进行了大量应用。虚拟现实技术的基础和核心是三维计算机图形学,分为四大模块:建模、渲染、动画、交互。目前国内大量的虚拟现实应用都局限于在西方开发的虚拟现实引擎的技术上进行开发的上层应用。我们这套丛书着重底层核心技术的讲解,三维计算机图形学在知识结构上来说需要数学、物理、工程、计算机编程、艺术五个方面。设计建模、渲染等算法需要微分几何、线性代数、概率统计等数学知识的理解和掌握;动画模拟需要流体、刚体等物理知识的理解和掌握;把这些数学、物理理论变为程序需要极强的编码能力,也就是从理论到实践的工程能力;三维图形学的最终表现形式是视觉上可看得到的,因此也需要良好的艺术修养和审美。虚拟现实和它所依赖的三维计算机图形学特别适合锻炼并能够融会贯通学生的数学、物理、工程、编程和艺术能力。三维计算机图形学是一个跨学科领域,三维图形学处理的是三维模型数据,学生在这个领域中学到的数学建模、工程等能力,也可以用到其他行业,如人工智能等,对其他行业的大数据进行分析和处理。

2008 年以来,全国各个高等院校纷纷在各自软件工程学科专业的基础上开设了数字媒体技术专业。数字媒体技术专业 and 计算机专业专业的区别是,前者主要是着重学习二维图像和三维图形相关的算法和应用开发,而后者还需要学习其他计算机科学相关的知识。由于开设和建立时间短,各学校的数字媒体技术专业的教学工作都还处在摸索阶段,也没有形成统一、成熟的教材体系。根据在数字媒体技术专业多年的教学实践经验,我们总结出本专业要以计算机三维图形学的理论和算法为基础,以三维应用开发为导向进行建设。

根据多年一线教学经验与反馈,以及当前的三维图形学研究成果,我们编写了本套丛书。本套丛书涵盖了三维图形学算法的三个方面:建模、动画和渲染。内容根据数字媒体技术专业的教学特点分散到 5 本 VR 三维技术系列图书中。通过本系列专业图书,再加上已有的成熟的计算机基础编程教材,以及三维软件使用的教材,就可以完整地覆盖数字媒体技术专业的所有课程。

书里的代码采用 C# 编程语言。C# 编程语言是一种结合了 C++ 和 Java 优点的编程语言。C# 语言相对于其他编程语言来说比较容易学习和掌握,但是本套丛书里讲述的原理和算法不仅限于 C# 语言。读者可以通过示例中的代码,采用自己熟悉的编程语言来进行编程。本套丛书包含了很多计算机图形学会议 Siggraph 论文里最新的、核心的、关键突破和进展的图形学算法讲解、实现和分析。

前 言

虚拟现实技术里面最核心的基础是计算机三维图形学技术。其中一个重要的模块是三维模型的变形技术。这个技术涉及如何光滑地改变一个现有模型的形状，从而使该模型的整体姿势发生变化，但是保持局部细节不变。例如，把一个杯子拉伸等。变形算法从 20 世纪 80 年代开始，到现在已经 30 多年的历程。有各种各样的类别和方法，在商业软件，如 Maya 等已经进行应用。

本书不仅讲述了这些商业系统中成熟的应用，而且还讲述了最新的、还没有集成到商业软件中的变形算法。虽然现在大部分虚拟现实的应用都是基于商业软件进行三维模型的功能实现，但是针对特定的需求，需要掌握具体底层的技术，从而进行开发。这本书针对变形算法提供了系统的讲解和实例，从而使读者能够对变形算法有全面的了解、学习和掌握，并且可以直接进行应用。

三维模型变形算法的设计也需要用到微分几何、物理等知识。但是它们是作为我们设计算法的基石和指导，在算法设计过程中，我们针对特定的需求，需要通过近似、简化等方法建立灵活的新的数学模型。也就是三维图形学的算法不仅仅是数学、物理的纯粹计算。

《三维模型变形算法：理论和实现（C#版本）》介绍了三维模型变形算法和线条抽取算法。本书分为 14 章，详细讲述了 Blender 软件中的变形、网格变形算法、外包框变形算法、均值坐标变形算法、格林变形算法、拉普拉斯变形算法、拉普拉斯矩阵在三维模型近似、光滑、优化、骨骼抽取、顶点之间最短距离算法上的应用；还有三维模型的频谱分析、骨骼动画算法、蒙皮算法、三维曲线生成算法、三维模型特征曲线的抽取算法。本书包含了三维模型处理的各种核心算法。

本书不仅可以作为数字媒体技术专业的专业基础课，还可以作为计算机学科和软件工程学科“数据结构和算法”、“计算机图形学”等课程的教材和参考书。需要书里部分代码的读者可以发邮件向作者索取，邮箱地址：graphicsresearch@qq.com。

赵 辉

2017 年 05 月于清华大学近春园



作者简介

赵辉，虚拟现实专家、清华大学丘成桐数学科学中心访问学者、哈佛大学访问学者。主要研究计算微分几何、拓扑、三维模型处理算法（三维模型简化、细分、分割、变形、光滑、参数化、向量场、四边形化等）、三维动画算法（骨骼动画、蒙皮算法）、渲染算法（非真实感渲染、实时渲染、基于物理渲染），以及三维技术在3D打印、虚拟现实、增强现实、三维游戏、手机游戏、影视特效等的应用。



顾险峰，师从国际著名微分几何大师丘成桐院士，现于纽约州立大学石溪分校计算机科学系和应用数学系终身教授，清华大学丘成桐数学科学中心客座教授，大连理工大学海天学者，首都师范大学数字几何和成像实验室主任等。2005年获得美国国家自然科学基金 CAREER 奖，2006年获得中国国家自然科学基金海外杰出青年学者奖，2013年第六届世界华人数学家大会晨兴应用数学金奖等。



顾险峰教授和丘成桐先生及其合作者共同创立了一门新兴的跨领域学科：计算共形几何。这门学科结合了现代几何和计算机科学，广泛应用于计算机图形学、计算机视觉、可视化、几何建模、网络和医学图像等领域。

雷娜，大连理工大学软件学院教授，博士生导师，北京市成像技术高精尖创新中心兼职研究员；中国工业与应用数学学会几何设计与计算专业委员会委员；中国数学会计算机数学专业委员会委员；美国数学会 Mathematical Review 评论员；清华大学数学科学中心访问教授；纽约州立大学石溪分校计算机系访问教授；德克萨斯大学奥斯汀分校计算工程与科学研究所 research fellow；中科院数学与系统科学研究院访问学者。主要研究兴趣是应用现代微分几何和代数几何的理论与方法解决工程及医学领域的问题，聚焦于计算共形几何、计算拓扑、符号计算及其在计算机图形学、计算机视觉、几何建模和医学图像中的应用。



目 录

第1章 Blender 软件中的变形	1
1.1 变形介绍	1
1.2 外包框变形	2
1.2.1 外包框变形步骤	2
1.2.2 外包框变形效果和分析	6
1.2.3 外包框变形实验	7
1.3 网格变形	9
1.3.1 网格变形步骤	9
1.3.2 网格变形方法效果和分析	12
1.3.3 网格变形效果	13
1.4 拉普拉斯变形	14
1.4.1 拉普拉斯变形步骤	14
1.4.2 拉普拉斯变形效果和分析	18
1.4.3 拉普拉斯变形实验	19
第2章 FFD 变形算法	21
2.1 FFD 介绍	21
2.2 FFD 算法数学推导	22
2.3 FFD 算法步骤	24
2.4 实现代码	24
第3章 均值坐标变形算法	27
3.1 均值坐标介绍	27
3.2 重心坐标	28
3.3 数学推导	30
3.4 变形步骤	32
3.5 效果分析	35
第4章 格林坐标变形算法	37
4.1 格林变形介绍	37
4.2 算法步骤和代码	38
4.3 其他外包框变形坐标	42
第5章 三维模型上的矩阵	44
5.1 邻接矩阵	44
5.2 组合拉普拉斯矩阵	50
5.2.1 拉普拉斯矩阵介绍	50

5.2.2 拉普拉斯矩阵构建	51
5.3 余切拉普拉斯矩阵	54
第6章 拉普拉斯变形算法	60
6.1 微分坐标	60
6.2 变形算法基础	63
6.2.1 变形介绍	63
6.2.2 数学模型构建	65
6.2.3 拉普拉斯变形算法代码	66
6.3 拉普拉斯变形迭代算法	69
6.3.1 迭代法介绍	69
6.3.2 数学模型构建	71
6.3.3 迭代法核心代码	72
6.4 ARAP 变形算法	75
6.4.1 算法思想	75
6.4.2 数学模型构建	76
6.4.3 ARAP 核心代码	78
第7章 拉普拉斯模型处理算法	81
7.1 三维模型近似算法	81
7.1.1 三维模型近似概述	81
7.1.2 数学系统构建和代码	82
7.1.3 近似模型算法效果图	83
7.2 拉普拉斯模型优化算法	87
7.2.1 三维模型优化介绍	87
7.2.2 数学模型构建	88
7.2.3 优化算法核心代码	90
7.2.4 优化算法效果	92
7.3 拉普拉斯光滑算法	93
7.3.1 光滑算法介绍	93
7.3.2 能量函数和数学系统	94
7.3.3 光滑算法核心代码	96
7.3.4 光滑算法效果展示	98
7.4 非奇异平均曲率流光滑算法	100
7.4.1 光滑算法分析	100
7.4.2 数学推导和核心代码	102
7.4.3 CMCF 光滑算法效果展示	104
7.5 骨骼抽取	109
7.5.1 骨骼抽取概述	109
7.5.2 数学模型构建和核心代码	109
7.5.3 骨骼抽取效果	113

第8章 三维模型频谱分析	116
8.1 矩阵的频谱	116
8.2 菲德尔向量	118
8.3 节点域	120
8.4 连通体和特征符	123
8.5 特征向量近似	125
8.5.1 数学原理	125
8.5.2 近似算法步骤	127
8.5.3 效果分析	129
第9章 顶点间最短距离算法	132
9.1 最短距离概念	132
9.2 Diffusion 距离算法	134
9.3 Commute Time 距离算法	137
9.4 双和谐距离	138
第10章 Blender 中的蒙皮技术	142
10.1 两个关节的简单蒙皮	142
10.2 仙人掌蒙皮	150
10.3 马匹蒙皮	155
第11章 骨骼动画算法	167
11.1 动作捕捉	167
11.2 BVH 文件格式	168
11.2.1 BVH 格式定义	168
11.2.2 文件实例	169
11.2.3 加载 BVH 文件代码	172
11.3 骨骼结构算法	176
11.3.1 骨骼结构	176
11.3.2 算法原理	177
第12章 蒙皮算法	187
12.1 概述	187
12.2 SMD 蒙皮文件	189
12.2.1 文件格式定义	189
12.2.2 文件加载	191
12.3 线性混合算法	201
12.4 对偶四元素数算法	205
12.4.1 数学概念	205
12.4.2 算法原理	206
12.5 DQS 和 LBS 对比	210
12.5.1 优劣性	210
12.5.2 测试模型生成	213

12.6 蒙皮显示	217
第13章 曲线	221
13.1 参数化曲线	221
13.2 贝塞尔曲线	227
13.2.1 概述	227
13.2.2 贝塞尔曲线公式	228
13.2.3 度数升级	229
13.2.4 贝塞尔曲线代码	231
13.2.5 贝塞尔曲面	235
13.3 B-Spline 曲线	236
13.3.1 B样条曲线特点	236
13.3.2 B-Spline 曲线公式	238
13.3.3 B样条曲线代码	239
13.4 NURBS 曲线	243
13.4.1 定义和属性	243
13.4.2 NURBS 曲线公式	245
13.4.3 NURBS 曲线代码	246
第14章 三维模型特征线条抽取算法	252
14.1 Blender FreeStyle	252
14.2 特征线条分类	261
14.3 剪影线	262
14.4 轮廓线	265
14.5 脊谷线	268
14.6 主观轮廓线	272
14.7 视脊线	274
14.8 高光线条	276
14.9 其他线条	278
参考文献	280

浅谈曲面变形的几何理论基础

在计算机图形学、计算机视觉、计算机辅助设计和医学图像等领域，曲面变形具有根本的重要性。曲面变形的计算方法非常丰富多彩，所用的理论也涉及现代数学的很多分支。这里，我们先简单介绍一些常见算法背后的基础理论。当然，从抽象理论到具体算法，也需要长期的探索和艰苦卓绝的努力。

1. 几何泊松方程

最为常见的曲面变形算法可以归结为求解曲面上的椭圆形偏微分方程（Elliptic Partial Differential Equation）。假设 S 是嵌入在三维欧氏空间中的曲面，自然带有诱导的欧氏黎曼度量 g ，我们可以选择局部等温坐标 (x, y) ，使得黎曼度量具有简单的公式：

$$g = e^{2\lambda(x,y)} (dx^2 + dy^2)$$

如图 1 所示。

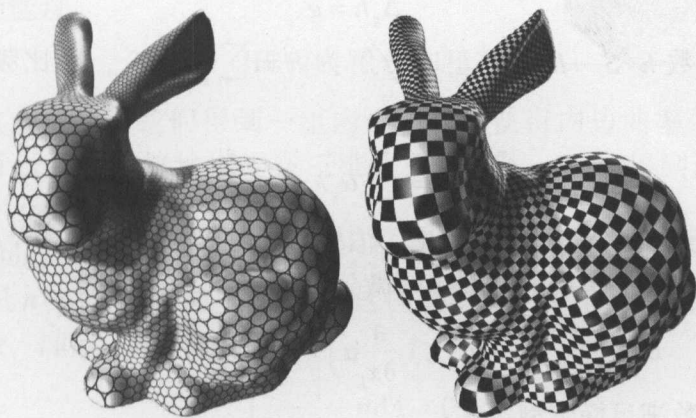


图 1 曲面的等温坐标， $g = e^{2\lambda(x,y)} (dx^2 + dy^2)$

曲面的 Laplace - Beltrami 算子定义为

$$\Delta g = \frac{1}{e^{2\lambda(x,y)}} \left(\frac{\partial^2}{\partial x^2} + \frac{\partial^2}{\partial y^2} \right)$$

Laplace 算子的特征根和特征函数满足：

$$\Delta_g f_k = \lambda_k f_k$$

这里特征根非负，构成曲面的谱：

$$0 = \lambda_0 \leq \lambda_1 \leq \lambda_2 \leq \dots \leq \lambda_n \leq \dots$$

其对应的特征函数为

$$\{f_0, f_1, f_2, \dots, f_n, \dots\}$$

这里第一个特征函数 f_0 为常数。曲面上所有平方可积函数构成了希尔伯特空间 (Hilbert Space), $H(S)$, 其内积定义为

$$\langle f, g \rangle = \int_S fg dA$$

我们可以证明特征函数满足的性质:

$$f_k - \operatorname{argmin}_{f \perp f_i} \langle f, f \rangle, \quad 0 \leq i \leq k-1$$

我们将特征函数进行缩放变换, 使得 $\|f_k\| = 1$ 。由此特征函数构成了希尔伯特空间的一组单位正交基底:

$$\langle f_i, f_j \rangle = \delta_{ij}$$

曲面上任意一个平方可积的函数 $g: S \rightarrow \mathbb{R}$, 都可以分解成基底的线性组合:

$$g = \sum_{k=0}^{\infty} \mu_k f_k$$

这里 $\{\mu_k\}$ 被称为是 g 的频域系数。这种分解方式本质上是黎曼流形上的傅里叶分解 (Fourier Transform)。如果底流形是单位圆周, 那么这种分解方式就是经典的傅里叶分解。

考察曲面上的泊松方程:

$$\Delta_g h = g$$

我们假设函数 $h: S \rightarrow \mathbb{R}$ 的傅里叶分解为 $h = \sum_{k=0}^{\infty} \alpha_k f_k$, 对比频率域系数我们得到:

$$\Delta_g \sum_{k=0}^{\infty} \alpha_k f_k = \sum_{i=0}^{\infty} \alpha_k \lambda_k f_k = \sum_{k=0}^{\infty} \mu_k f_k$$

由此我们得到 $\alpha_k = \mu_k / \lambda_k$, 这给出了泊松方程解的公式。

更为一般的, 我们考察散度型椭圆微分算子:

$$Lu - \sum_{i,j} \frac{\partial}{\partial x_i} \left(a^{ij}(x) \frac{\partial}{\partial x_j} u \right) + \sum_j b^j(x) \frac{\partial}{\partial x_j} u + cu$$

这里二次系数满足椭圆型条件:

$$c|\xi|^2 \leq \sum_{i,j} a^{ij}(x) \xi_i \xi_j \leq C|\xi|^2, \quad \forall \xi \in \mathbb{R}^2$$

这里 c, C' 为正数。那么, 可以证明, 存在曲面的另外一个黎曼度量 \tilde{g} , 在 \tilde{g} 下椭圆型偏微分方程被转化为经典的泊松方程, 因此可以用类似方法解出。这里黎曼度量 \tilde{g} 可以由矩阵 $(a^{ij}(x))$ 构造拟共形映射来得到。

2. 等距嵌入

Weyl 问题: 曲面变形的一种途径是改变曲面的黎曼度量, 由此改变曲面在三维欧氏空间中的嵌入, 最为知名的是 Weyl 问题和 Minkowski 问题, 如图 2 所示。Weyl 在 1915 年提出了如下的问题, 在拓扑球面上给定一个黎曼度量, 使得高斯曲率处处为正, 求如何在三维欧氏空间中实现这个黎曼流形? Alexandrov 和 Pogorelov 给出了这

个问题完整答案。

在离散情形下, Weyl 问题提法如下, 给定一族平面多边形, 给定这些多边形的粘和方式, 即哪两条边应该粘在一起。如果所有顶点处的离散高斯曲率非负, 即每个顶点处所有顶角之和不大于 2π 。那么, 我们能够将这组多边形粘和成三维空间中的一个凸多面体。从物理直觉上, 我们可以手工搭建这个凸多面体, 粘贴多边形的次序应该保持多面体的稳固性并且无歧义性, 那么我们可以顺利实现凸多面体的嵌入。但是, 我们需要一个严密的计算方法。

首先, 我们可以从组合上对多面体进行四面体三角剖分, 得到四面体网格 M 。 M 的边界度量已知, 所有内边边长为自由变量。我们为所有内边赋予一个初始边长, 从而得到一个初始欧氏度量。每条内边 $|v_i, v_j|$, 我们定义离散曲率:

$$K_{ij} = 2\pi - \sum_{k,l} \theta_{ij}^{kl}$$

这里 θ_{ij}^{kl} 为在四面体 $|v_i, v_j, v_k, v_l|$ 内, 边 $|v_i, v_j|$ 上的二面角。我们定义离散希伯特-爱因斯坦能量:

$$HE(I) := \sum_{i,j} K_{ij} l_{ij}$$

通过优化这个能量, 我们得到一组内边边长, 使得内边曲率处处为 0。这样, 所有四面体就可以严丝合缝地堆砌在三维欧氏空间中, 从而得到凸多面体的等距嵌入。

Minkowski 问题: 假设 S 是三维欧氏空间中的一张封闭凸曲面, 给定任意一点 $p \in S$ 处的法向量 $\mathbf{n}(p)$ 和高斯曲率 $K(p)$, 这样就给出了一个定义在单位球面上的正值函数 $K: S^2 \rightarrow \mathbb{R}$, 满足:

$$\int_{S^2} \frac{1}{K(\mathbf{n})} \mathbf{n} dA_{\mathbf{n}} = 0$$

那么, 曲面 S 可以被重建出来, 所有的解彼此相差一个平移。

在离散情形, 如图 2 所示, 给定一个凸多面体每个面的法向量 \mathbf{n}_i , 和面积 A_i , 满足条件:

$$\sum_i A_i \mathbf{n}_i = 0$$

那么存在一个凸多面体 M , 其各个面的法向量和面积等于给定值, 并且这种凸多面体彼此相差一个平移。

其计算方法如下: 对于每个面, 我们构造平面方程 $\pi_i(p) = \langle p, \mathbf{n}_i \rangle - h_i$, 这些平面构成凸多面体 $M(h)$ 。我们极大化多面体 $M(h)$ 的体积, 满足约束条件 $\sum_i h_i A_i = 1$, 极值点存在并且唯一, 给出了 Minkowski 问题的解。

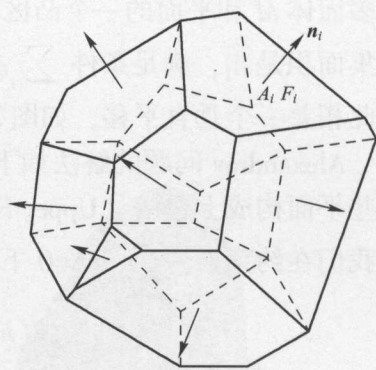


图 2 Weyl 问题和 Minkowski 问题

Alexandrov 问题: Alexandrov 问题和 Minkowski 问题非常类似, 给定一个开放的凸多面体 M 和平面的一个凸区域 Ω , 每个面的法向量 \mathbf{n}_i 给定, 每个面的投影和 Ω 的交集面积是 A_i , 满足条件 $\sum_i A_i = \text{Area}(\Omega)$, 则凸多面体存在, 并且这样的凸多面体彼此相差一个垂直平移, 如图 3 所示。

Alexandrov 问题的解法如下: 每一个面对应一个平面方程 $\pi_i(p) = \langle p, \mathbf{n}_i \rangle + h_i$, 这些平面构成上包络 (Upper Envelope) $U(h)$, $U(h)$ 每个面的投影面积记为 $w_i(h)$, 则我们在约束条件 $\sum_i h_i = 0$ 下优化下面的体积能量:

$$V(h) := \int^h \sum_i (A_i - w_i(\eta)) d\eta_i$$

则体积能量全局为凸, 最优解存在并唯一, 所得即为 Alexandrov 问题的解。

3. 共形形变

对应非凸的封闭曲面, 单单黎曼度量无法唯一决定曲面在三维欧氏空间中的嵌入, 我们需要给出更多的信息。例如, 曲面的平均曲率。

一种方法是将曲面嵌入在四元数空间 H 中。一个四元数 (quaternion) 表示为

$$\mathbf{q} = a + bi + cj + dk$$

其乘法规则由图 4 给出。四元数的共轭定义为

$$\bar{\mathbf{q}} = a - bi - cj - dk$$

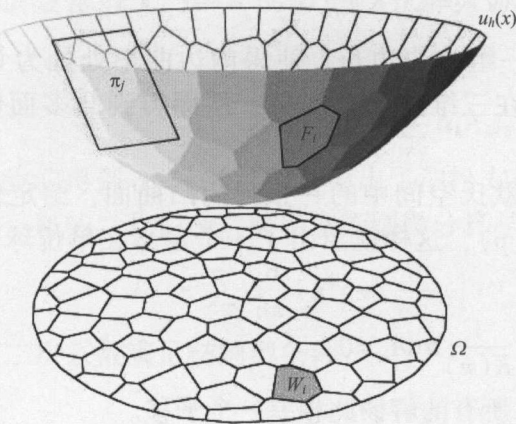


图 3 Alexandrov 问题

\mathbf{q}	1	i	j	k
1	1	i	j	k
i	i	-1	k	$-j$
j	j	$-k$	-1	i
k	k	j	$-i$	-1

图 4 四元数的乘法规则

四元数的模定义为 $|\mathbf{q}|^2 = \mathbf{q} \bar{\mathbf{q}}$, 由此, 我们得到四元数的逆为 $\mathbf{q}^{-1} = \bar{\mathbf{q}} / |\mathbf{q}|^2$ 。由此, 所有的四元数构成一个域 (Field)。

我们考虑三维空间中的一个旋转, 旋转轴为单位向量 (x, y, z) , 旋转角为 θ , 这个旋转可以被表示成一个四元数 (quaternion)

$$\mathbf{q} = \cos \frac{\theta}{2} + (xi + yj + zk) \sin \frac{\theta}{2}$$

我们将三维欧氏空间 E^3 嵌入在虚四元数 $\text{Img}H$ 集合中。给定一个向量 $\mathbf{v} \in \text{Img}H$,

旋转之后的向量表示为：

$$\tilde{v} = q^{-1} v q$$

因此，旋转加上放缩变换应该是 $\tilde{v} = \bar{q} v q$ 。

Spin 变换如图 5 所示。

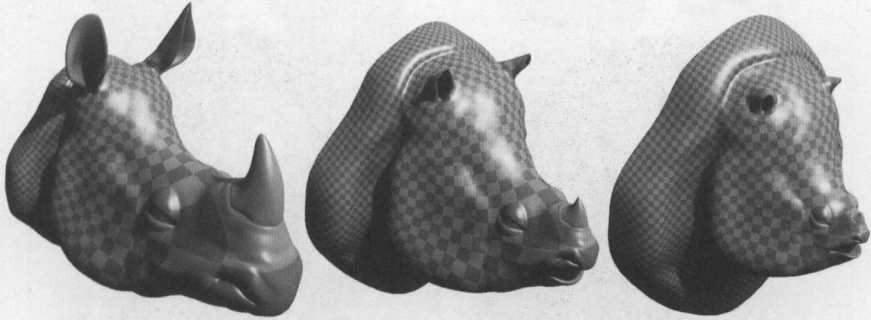


图 5 Spin 变换 (Crane, Pinkall and Schroder, Spin Transformations of Discrete Surfaces, ACM TOG 2011)

假设 S 是一个拓扑球面，在三维欧氏空间中两个浸入 (immersions), $f, \bar{f}: S \rightarrow \text{Img}H$, 彼此之间相差一个共形变换，那么固定曲面上任意一点 $p \in S$, 导映射 $df, d\bar{f}: T_p S \rightarrow \text{Img}H$ 相差一个相似变换，即相差一个 $\text{Img}H$ 中的旋转加放缩，由此我们得到基本方程：存在一个以四元数为值域的函数 $\lambda: S \rightarrow \text{Img}H$, 满足如下方程：

$$d\bar{f} = \lambda df$$

对上式两边同时求外微分，我们得到

$$0 - d(\bar{\lambda} df) - d\bar{\lambda} \wedge df - \bar{\lambda} df \wedge d\lambda = -2\text{Im}(\bar{\lambda} df \wedge d\lambda)$$

这意味着 $\bar{\lambda} df \wedge d\lambda$ 是一个实的 2-形式。进一步计算表明：

$$\lambda df \wedge d\lambda = \rho |df|^2$$

这里 ρ 和平均曲率的变化有内在联系：

$$\bar{H} |d\bar{f}| = H |df| + \rho |df|$$

由此，我们得到偏微分方程：

$$D\lambda = \rho\lambda$$

这里微分算子 D 被称为四元数狄拉克算子：

$$D\lambda := -\frac{df \wedge d\lambda}{|df|^2}$$

在实际应用中，给定嵌入在欧氏三维空间中一个初始曲面 $f: S \rightarrow E^3$, 用户定义平均曲率的变化 $\rho: S \rightarrow R$, 然后求解特征问题：

$$(D - \rho)\lambda = \gamma\lambda$$

这样，所得以四元数为值域的一对函数 $(\lambda, \rho + \gamma)$ 满足可积性条件，我们可以得到新的曲面 $\bar{f}: S \rightarrow E^3$ 和初始曲面共形等价。

对于高亏格曲面，这种方法无法保证是嵌入，目前依然在探索阶段。

4. 背景空间形变

另外一种曲面变形的方法是将曲面所在的背景空间整体形变，从而使得曲面跟随背景空间发生形变。如图 6 所示，我们将图片贴在一个样条曲面上，然后通过改变样条的控制点，使背景曲面形变，从而使得图像变形。



图 6 Free-form deformation

最优传输映射：一种通用的空间变形的方法是基于最优传输理论。给定三维空间中的一个凸区域 $\Omega \subset E^3$ ，我们在上面定义两个概率测度 μ, ν ，满足 $\int_{\Omega} d\mu = \int_{\Omega} d\nu$ ，那么存在一个凸函数 $u: \Omega \rightarrow \mathbb{R}$ ，其梯度映射 $T(p) := \nabla u(p)$ 满足以下两个条件。

(1) 将概率测度 μ 映成概率测度 ν ，这意味着对于一切可测集合 $E \subset \Omega$ ， $\mu(T^{-1}(E)) = \nu(E)$ ；

(2) 映射极小化传输代价：

$$E(T) := - \int_{\Omega} |p - T(p)|^2 d\mu(p)$$

这种映射被称为最优传输映射。

最优传输映射可以通过解蒙日 - 安培方程 (Monge - Ampere) 求得，凸函数 u 满足：

$$\det \left(\frac{\partial^2 u}{\partial x_i \partial x_j} \right) (p) = \frac{\mu(p)}{\nu \circ \nabla u(p)}$$

这种类型的 Monge - Ampere 方程的几何解法和 Alexandrov 问题的解法是类似的，都可以通过变分法求得。如图 7 所示，我们将膝盖骨骼模型嵌入在标准立方体，然后用最优传输映射将立方体变形，从而诱导膝盖骨骼的变形（绿色圆圈内区域）。

流体力学：另外一种常用方法是基于流体力学。我们在区域 Ω 内设计矢量场 ν ，定义了流场的速度场。每一个粒子的流动速度由 ν 给出，所有的流线都和速度场处处相切。这样，我们得到了单参数微分同胚群， $\varphi_t: \Omega \rightarrow \Omega$ ，满足常微分方程：

$$\frac{d\varphi_t(p)}{dt} = v(\varphi_t(p))$$

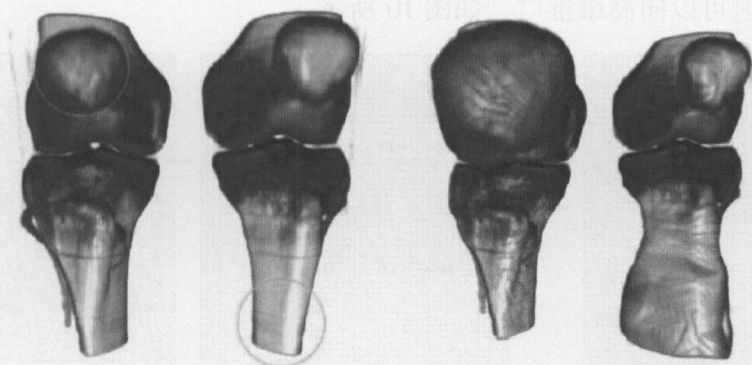


图7 背景空间的形变诱导曲面形变 (苏科华作)

给定 Ω 的任意微分形式 ω , 其关于 v 的李导数满足嘉当公式 (Cartan's magic formula):

$$L_v \omega = \frac{d\varphi_t^* \omega}{dt} - i_v d\omega + di_v \omega$$

由此, 我们可以通过设计矢量场, 控制体积元的变化, 进一步得到背景空间的微分同胚。

拟共形映射方法 (quasi-conformal mapping) 也经常被采用。如图8所示, 这种方法将无穷小圆映成无穷小椭圆。在每一点 p , 无穷小椭圆的偏心率和长轴方向可以用一个复数 $\mu(p)$ 来表示, 映射满足 Beltrami 方程:

$$\frac{\partial \varphi(p)}{\partial \bar{z}} = \mu(p) \frac{\partial \varphi(p)}{\partial z}$$

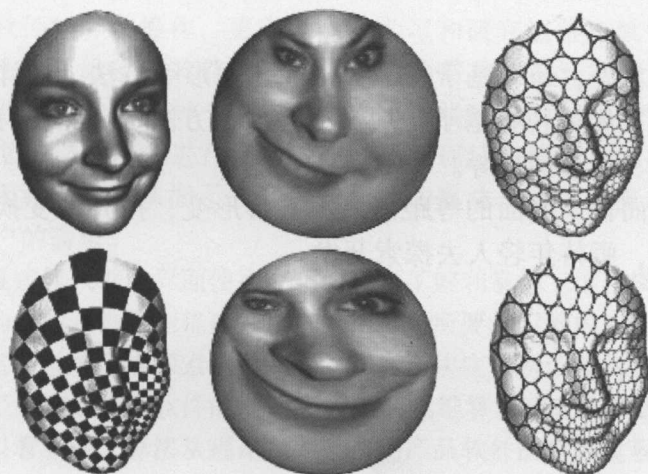


图8 拟共形映射

图 9 显示了用拟共形映射的方法求得的平面区域的微分自同胚，将给定的特征点对齐。这个微分同胚具有高度的扭曲，充分显示了这种方法的普适性和数值稳定性。拟共形映射可以向高维推广，如图 10 所示。

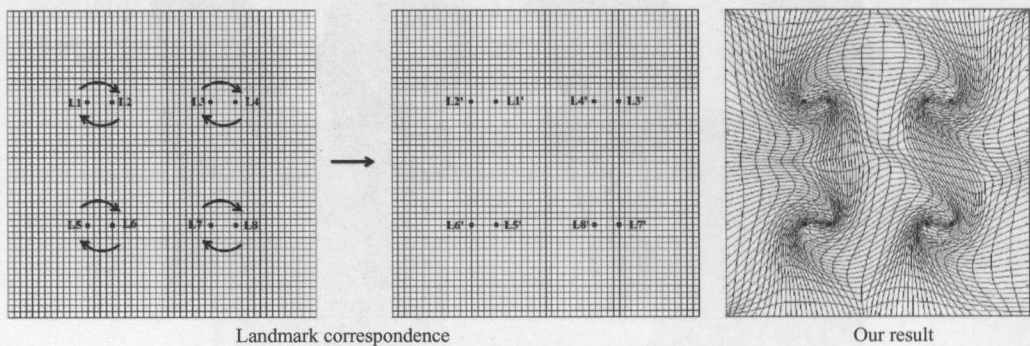


图 9 平面区域的拟共形映射 (Ronald Lui 教授作)

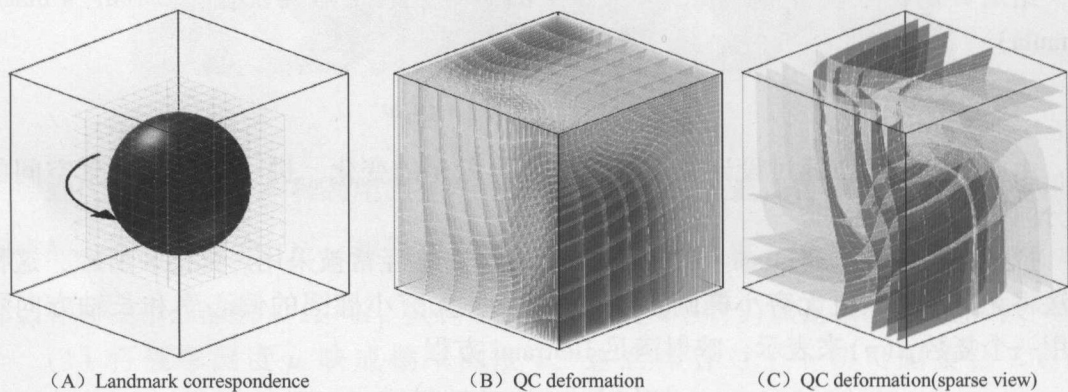


图 10 体区域的拟共形映射 (Ronald Lui 教授作。)

5. 小结

除了几何方法之外，还有基于物理模拟的曲面形变方法。在计算力学和计算机辅助制造 (CAE) 等领域，有基于物理模型的严密方法。自然也有基于数据驱动的方法，如动作捕捉、表情捕捉等。

单就几何方法而言，曲面的等距嵌入、共形形变，拟共形变换等领域，依然有大量的未解决问题，亟待年轻人去探索开发。

第 1 章

Blender 软件中的变形



1.1 变形介绍

当前有很多三维软件，其中一些是商业软件，如 3dsMax, Maya, Modo 等。还有一些是开源软件，如 Blender。三维软件有很多功能模块，这些功能模块可以分为三大类：建模 (Modeling)、动画 (Animation)、渲染 (Rendering)。本书主要讲述建模相关功能所依赖的计算机软件算法。三维模型的建模指的是通过软件制作三维模型，也就是通过点、线、面生成各种各样的三维模型，还可以对这些模型进行各种各样的操作，如光滑、细分、变形、求最短距离、着色、贴图 etc。这些建模的操作通过三维软件的菜单和键盘生成，但是菜单提供的只是一个界面，具体的实现还需要相关的计算机软件算法，也就是用户通过菜单或者键盘界面触发相应的算法实现建模的功能。

对于数字艺术家，在创作的时候，不需要知道三维建模所依赖的具体算法是如何实现的，只需要熟练掌握三维模型的界面使用就可以完成三维模型建模的创作。对于数字艺术家来说，关键的问题是如何通过现有的三维建模软件创作出有艺术效果的三维模型。三维软件就是数字艺术家的工具，与画布、画笔是画家的工具类似。数字艺术家的目标是通过三维建模软件工具创作，而画家的目标是通过画布和画笔进行绘画创作。但是对于从事计算机软件学习、开发和研究的科学家，工程师、教师和学生来说，不仅仅需要学习掌握三维建模软件的界面操作，更重要的是学习和研究这些三维建模软件底层所依赖的相关的三维算法。三维建模软件都是根据用户的需求进行开发的，但是用户的需求是持续更新、千变万化的。而且对于特定的需求，这些通用的三维软件，并不一定能够满足需求，需要根据特定的需求进行定制，如开发实现特定建模功能的插件，或者专门独立的应用程序。因此，只有掌握了三维软件相关的算法，才可以更加灵活地开发出相应的功能，满足用户的需求。

本书通过对三维建模软件的界面使用介绍使读者了解和掌握三维建模软件的特定模块是如何实现建模功能的，使不熟悉三维建模软件的读者对所学习的内容有感性的、初步的认识。在此基础上，然后继续深入探索这些相应的建模模块底层所依赖的算法的原理，以及实现的过程。通过已经成熟的建模软件的使用和底层的三维算法理论与实践的互相对比学习，可以加深理解，可以看到一个算法从理论到实践、到产品软件的整套过程。

在各种各样的三维建模软件中，Blender 是一个开源的软件，也就是软件的代码公开。并且可以在 Windows 等多平台上运行。Blender 软件不仅仅涉及建模，还有动画、材质、渲染、视频处理等模块和功能，并且可以用 Python 脚本语言进行脚本编程。支持 cycle、vray

等各种物理真实感渲染器，同时还自带有游戏引擎。也就是说 Blender 软件是一套三维应用完整的解决方案，可以用来制作各种各样的三维应用，如三维游戏、影视特效、三维可视化等。

虽然各种三维建模软件的基本功能都类似，但本书选用 Blender 作为展示模型处理功能的软件，是因为 Blender 软件是开源的，在掌握了本书介绍的各种模型处理的原理之后，就可以在 Blender 软件的源代码上进行开发，或者进行脚本编程。同时，Blender 作为开源的软件，也实现了很多最新的三维模型处理的算法成果。相比于商业软件来说，Blender 增加了很多最近几年计算机科学家研究出来的比较成熟的新功能。例如，对于三维模型的变形来说，大部分商业软件都具有网格变形和外包框变形，但是只有 Blender 把最新的变形技术拉普拉斯变形引入到了软件中。从而 Blender 的开源特性，以及具有新功能的特点非常适合于学生的学习，以及教师的研究，使学生可以更深入地了解软件的底层原理和接触最新的三维技术。而商业软件如 3dsmax，Maya 等的所有功能都是封闭的，外部无法了解这些软件内部的实现过程。

一个三维模型在确定之后，常常需要改变这个三维模型的形状，而不是从零开始重新建立一个不同形状的三维模型。改变现有三维模型形状的过程就称为三维模型的变形。例如，在图 1-1 中把一个人经过变形后可以变为不同姿势的人。

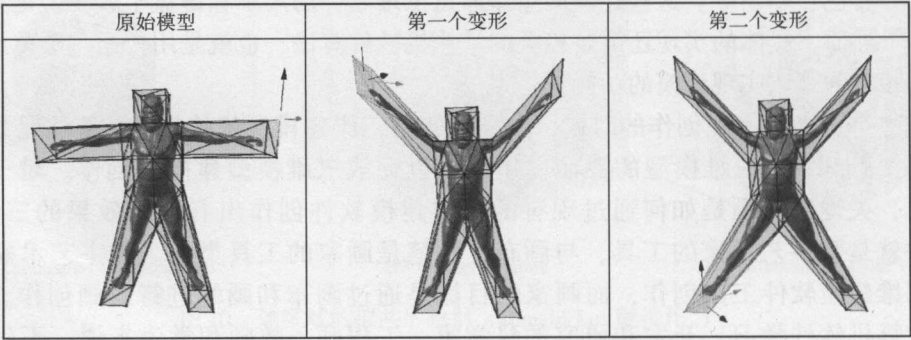


图 1-1 三维模型变形

三维模型的变形功能和操作是三维模型处理中非常重要的一个环节。通过这个变形操作可以极大地提高建模效率，而不需要每个不同姿势的模型都重新开始制作。本章主要介绍在 Blender 软件中如何使用三维模型变形功能，并展示各种实验效果。在 Blender 软件中主要有 3 种方式的变形方法：外包框变形（Mesh Deform）、网格（Lattice）变形和拉普拉斯变形（Laplacian Deform），这 3 种方法各有优劣。



1.2 外包框变形

1.2.1 外包框变形步骤

Blender 里外包框变形功能的基本方法是通过给需要变形的三维模型外面附加一个包围框，然后通过改变外包框的形状，从而相应地改变外包框内部包含的三维空间，然后三维空

间的改变引起三维空间里内嵌的三维模型的形状的改变。三维模型的变形虽然也可以通过改变三维模型上每个顶点的位置来得到变形的效果,但是一个三维模型通常有几百个、上千个顶点,改变每个顶点的位置费时费力,实现起来非常困难。而外包框通常只有很少的几个顶点,从而可以通过改变外包框上顶点的位置来改变外包框的形状,从而间接地改变外包框所包含的很多内部顶点数三维模型的形状。

Blender 软件外包框变形方法的步骤大致可以分为导入外包框、导入要变形的三维模型、将外包框和三维模型进行绑定、改变外包框顶点的位置、改变外包框的形状、改变三维模型的形状这几步。

第一步:在 Blender 软件打开的时候,主要场景界面中已经存在一个默认的立方体(Cube)物体。可以用这个默认的立方体作为变形的外包框控制网格,也可以使用自己导入的其他三维模型作为外包框控制网格。

第二步:通过 Blender 的菜单界面可以导入一个 obj 格式的将要进行变形的模型,如导入一个兔子三维模型。导入模型的界面如图 1-2 所示。

第三步:为了便于观察和展示,需要对三维模型进行一些调整。首先将场景中默认的立方体模型改为网格(Wire)显示模式,这样就可以透过网格的外包框看到内部要变形的三维模型,如图 1-3 所示。

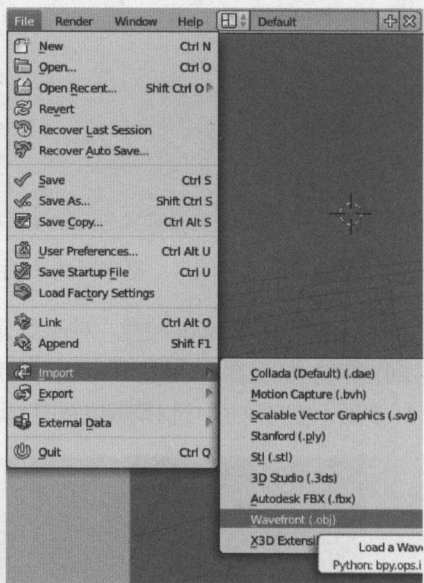


图 1-2 导入模型

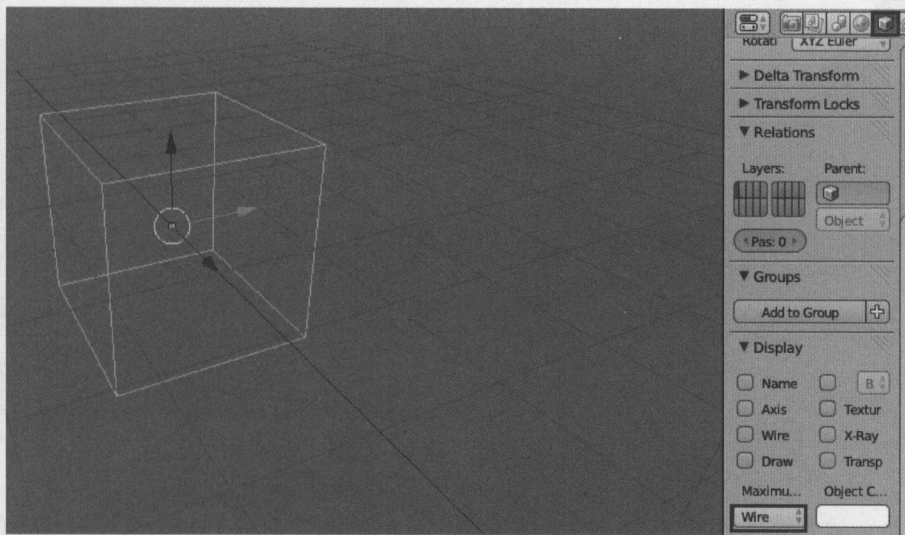


图 1-3 改变物体显示模式

第四步:选中导入的兔子(bunny)模型,调整其缩放大小和位置,使兔子模型位于立方体外包框的内部,如图 1-4 所示。

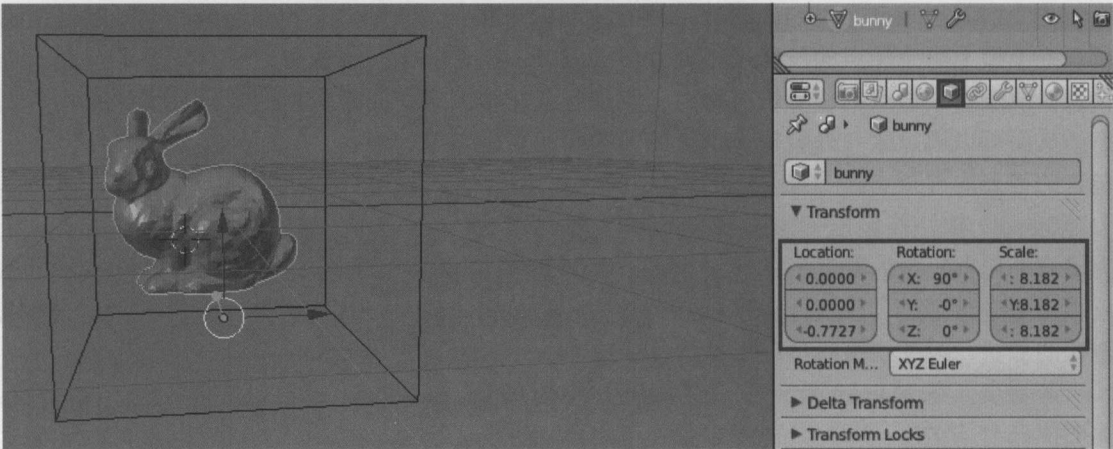


图 1-4 调整物体属性

第五步：在 Blender 中，对于给定的三维模型，除了变形外，还有各种各样的操作，这些操作称为修改器。为兔子模型添加一个变形修改器（Modifier）来对其进行变形，在如图 1-5 所示的界面中选择外包框变形（Mesh Deform）修改器。

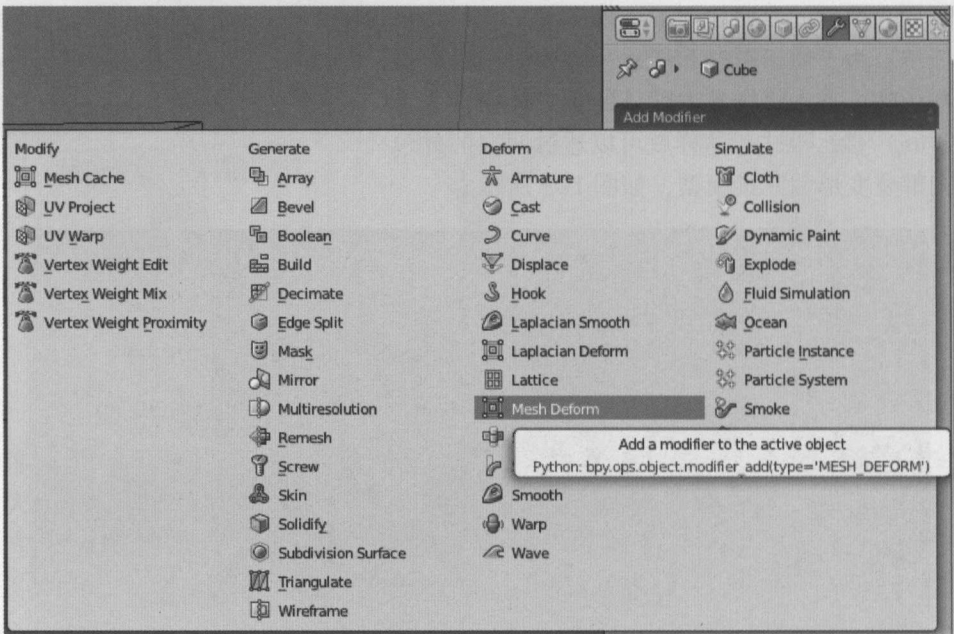


图 1-5 为模型添加外包框变形（Mesh Deform）修改器（Modifier）

第六步：这一步是绑定操作，绑定操作把三维模型和外包框关联起来。虽然在前几步把三维模型放到立方体外包框里面，但是这只是确定了几何位置，还需要通过绑定操作逻辑上确定三维模型所关联的特定的外包框。在如图 1-6 所示界面中的 Object 下拉框中选择立方体（Cube）模型作为外包框控制网格，然后单击 Bind 按钮将这个立方体外包框网格和要变形的兔子模型进行绑定。

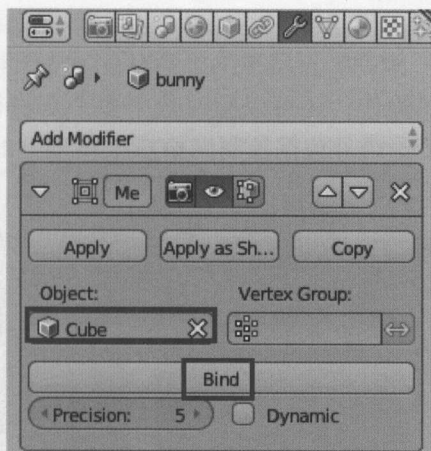


图 1-6 在 Blender 软件中为模型绑定控制网格

第七步：进入立方体外包框模型的编辑模式（Edit Mode）。进入编辑模式的方法可以通过单击右上方物体后面的图标，也可以在下方的下拉框中选择模式，如图 1-7 所示。在编辑模式下才可以改变三维模型顶点的位置，对三维模型进行编辑。

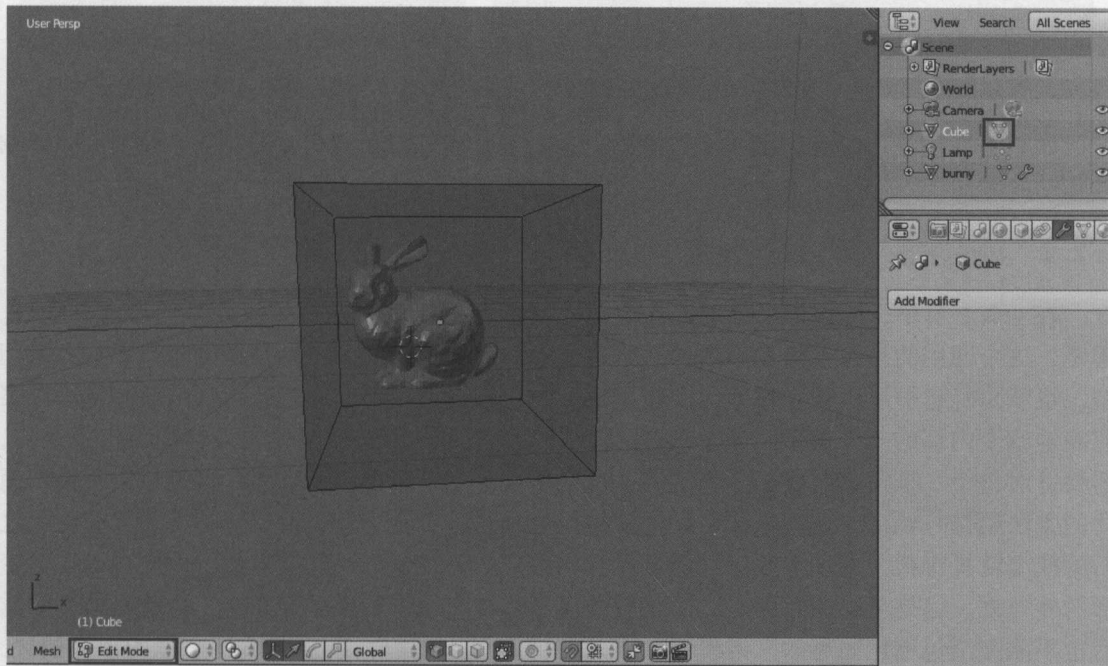


图 1-7 选择编辑模式

第八步：选中外包框上的任意一点，然后用右键拖动立方体外包框控制网格上选中的点到其他位置再松开，就可以改变外包框的形状。改变外包框上每个点的位置，就会改变外包框的最终形状。从而外包框所包围的三维模型的形状就会发生改变。

第九步：随着鼠标的移动，可以把外包框上的控制点拖动到满意的位置后单击鼠标左键确定，或者单击鼠标右键取消这次拖动的操作，如图 1-8 所示。另外，也可以通过右

键选择外包框控制网格上的某一点，然后用鼠标左键拖动该点上的坐标轴进行固定方向的移动。

第十步：通过这种方式的操作，还可以同时选中多个点进行相同的位移，只需要按住 Shift 键，然后可以右键选择若干个外包框控制网格三维模型上的点，然后左键拖动坐标轴，对这些点进行移动，如图 1-9 所示。

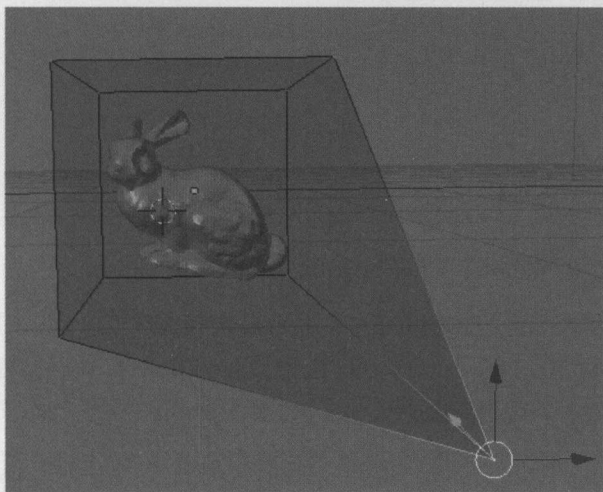


图 1-8 使用外包框变形方法对模型进行变形

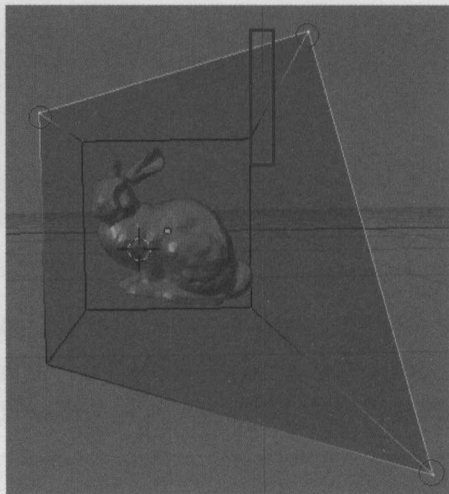


图 1-9 多点同时位移对模型进行变形

如图 1-9 所示，被标记的 3 个点被同时选中了，而标记的蓝色直线则表示这 3 个点沿 Z 轴移动的方向。

1.2.2 外包框变形效果和分析

在 1.2.1 节中，外包框采用的是立方体，立方体可以作为任意三维模型变形的外包框。但是，这个通用的外包框的变形效果不是最优的。假如外包框和三维模型的外形形状大致相似，也就是外包框可以很紧贴地包围住要变形的三维模型，那么通过外包框的改变就可以很直观地改变内部的三维模型。如图 1-10 所示的各种不同的三维模型的外包框。也就是说，需要针对每个三维模型制作一个特定的、专属于这个三维模型的外包框。通常可以用模型简化的方法制作外包框，也就是给定一个三维模型，然后对这个三维模型进行简化，简化完之后的模型就具有很少的顶点数量，但是形状和原来的三维模型还比较近似，因此可以作为变形的外包框。但是值得注意的是，简化后的模型往往和原有的模型会相交，也就是即使缩放后也不能完全包围住原有的三维模型，因此在绑定操作之前，需要调整外包框的顶点位置，使外包框可以完全包围要变形的三维模型。外包框变形可以很直观地改变三维模型。改变外包框上某个顶点的位置，通常指影响这个顶点附近的三维模型上的顶点位置。例如，在图 1-10 中第二行第二列的仙人掌三维模型上，右边淡黄色的是被选择的外包框顶点，改变这些顶点的位置，只改变距离这些顶点比较近的仙人掌三维模型上相关顶点的位置，而仙人掌的另一个分支由于距离外包框正在移动的控制顶点位置比较远，因此几乎不受影响。外包框变形的这个特点使得这个方法具有很直观的变形效果，也就是变形的结果和用户的预期比较一致。

图 1-10 展示了各种不同形状的三维模型采用外包框进行变形的效果。从中可以看出，通过外部的外包框，各种不同形状的目标变形结果都可以实现。

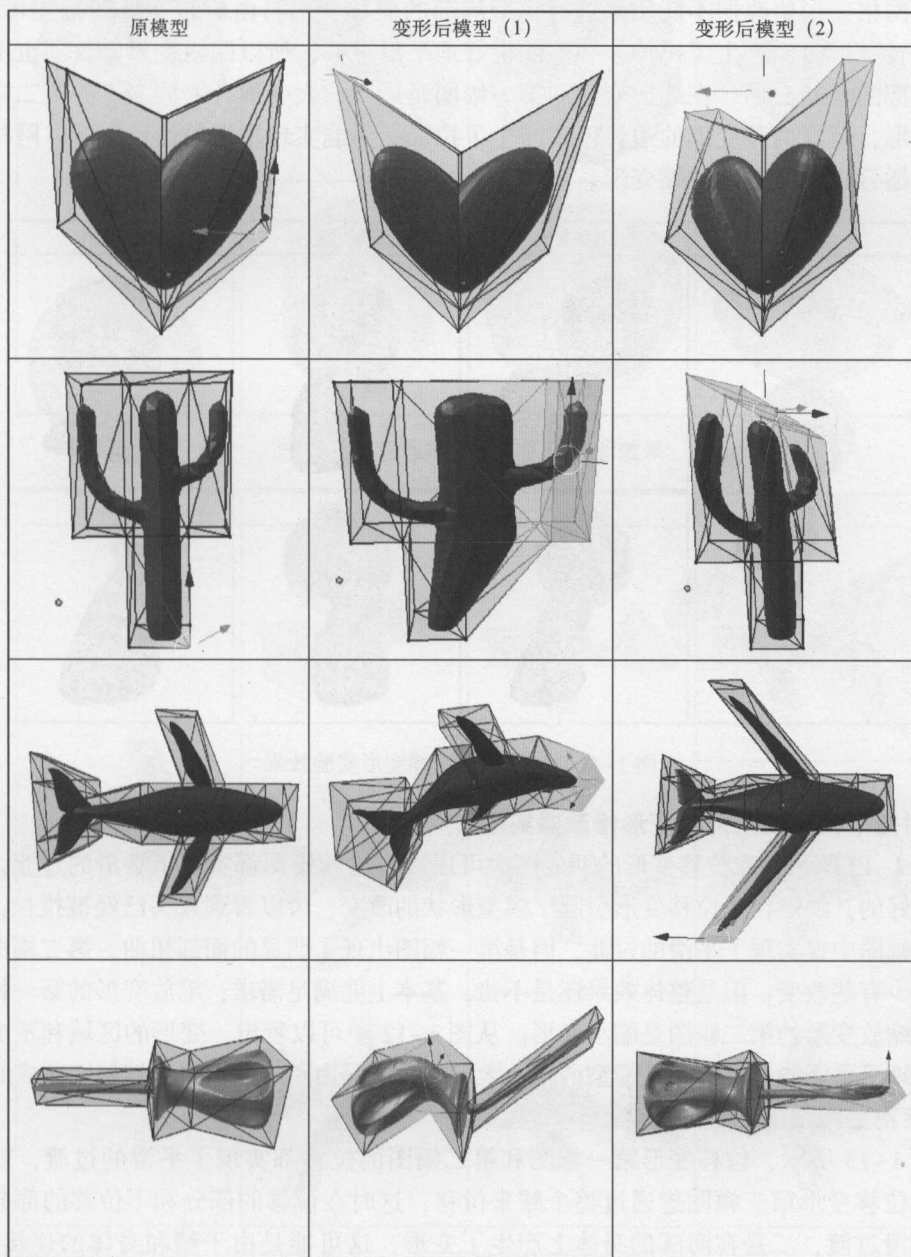


图 1-10 外包框变形效果

1.2.3 外包框变形实验

本小节通过在各种三维模型上对三维模型的一部分和整个三维模型进行位移、旋转、缩放等变形。通过这些不同设置、不同方法的效果分析和对比，可以更加深入了解外包框变形方法的特点。

1. 只对模型的一部分用网格变形修改器实验

如图 1-11 所示，从位移变形的两幅图可以看出，位移时产生了拉伸，不能实现平滑变形，说明网格变形修改器不能用来进行模型局部的位移变形；由旋转变形两幅图可以看出，在进行旋转变形时也产生了拉伸，并不能很好地平滑过渡，所以网格变形修改器也不适合用来进行局部的旋转变形；在缩放变形的第一幅图是局部放大变形，缩放变形的第二幅图是局部缩小变形，但这两种变形的边界位置产生了拉伸，不能实现平滑的过渡。所以网格变形修改器也不适合进行局部的放缩变形。

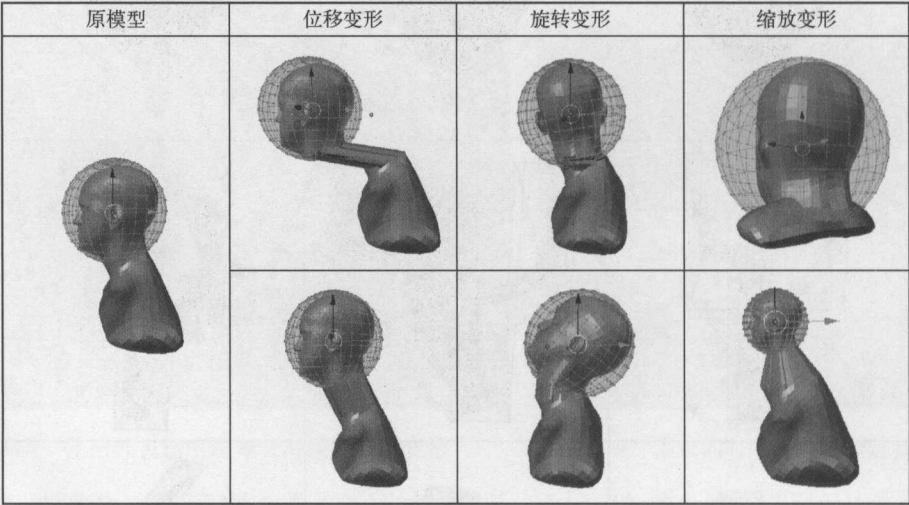


图 1-11 三维模型局部变形实验效果

2. 对整个模型使用网格变形修改器实验

如图 1-12 所示，在位移变形的两幅图中可以看出这两幅图都实现了平滑的过渡，变形效果还是很好的，缺点在于位移变形引起了模型形状的改变，可以看到人头已经被拉长；在旋转变形的两幅图中也实现了平滑的过渡，但是第一幅图出现了明显的面部扭曲，第二幅图的模型形状也多少有些改变，但是整体效果还是不错的，基本上能满足需求；缩放变形的第一幅图是放大变形，缩放变形的第二幅图是缩小变形。从图 1-12 中可以看出，变形的区域和不变形的区域之间实现了平滑的过渡，但是模型的扭曲太严重，这是由于控制网格不够精细造成的。

3. 使用更精细的网格进行变形

如图 1-13 所示，位移变形第一幅图和第二幅图的位移都实现了平滑的过渡，效果非常好，但是位移变形第三幅图想通过整个鳍来位移，这时在位移的部分和不位移的部分的交界处没有平滑过渡，二是在海豚的身体上产生了变形，这可能是由于鳍和身体的连接比较少，不便于平滑过渡引起的；旋转变形的第一幅图是尾巴向上旋转的变形，可以看出变形很平滑，效果很好，旋转变形的第二幅图是尾巴向外旋转 90° 的变形，效果也很好，过渡很平滑，旋转变形的第三幅图是尾巴向外旋转 180° 的变形，在变形部分和不变形部分的交接处出现了打结现象，没能将旋转继续向前传递；缩放变形的第一幅图是对海豚尾巴放大的变形，缩放变形的第二幅和第三幅图都是尾巴缩小的变形，由缩放变形的三幅图可以看出变形的效果很好，过渡很平滑，模型没有出现太明显的形变。

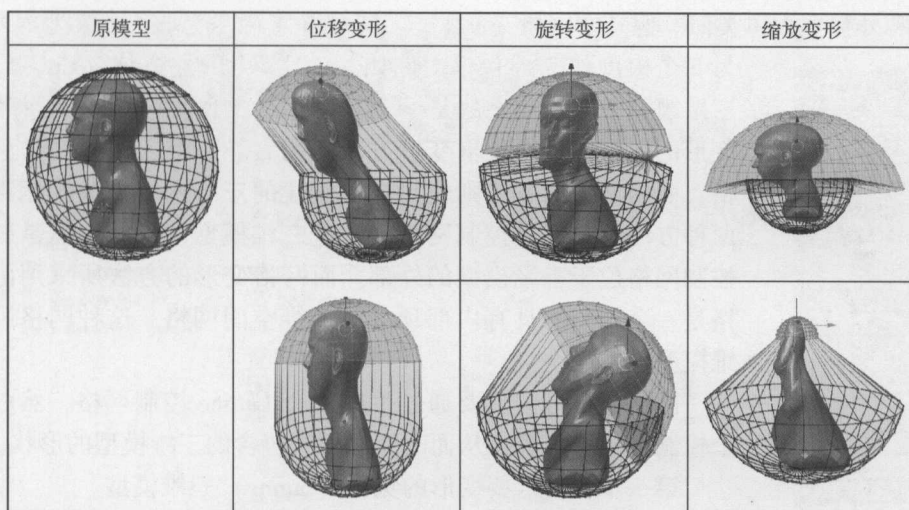


图 1-12 三维模型整体变形实验效果

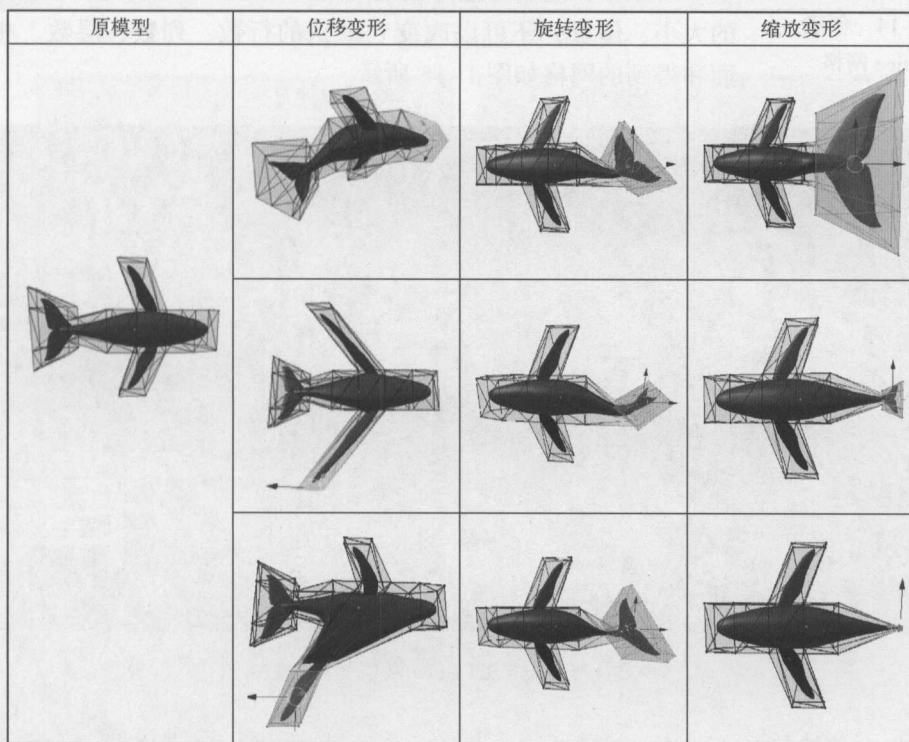


图 1-13 三维模型变形实验



1.3 网格变形

1.3.1 网格变形步骤

除了外包框变形方法外，Blender 软件还具有网格（Lattice）变形的方。网格变形的

操作过程和外包框大体类似。但是在网格变形中，不需要除了待定变形三维模型本身以外的另一个模型作为外包框。例如，1.2 节中的立方体（Cube）外包框三维模型，取而代之的是一个自动生成的网格。网格变形和外包框变形的区别在于外包框变形方法使用另外一个三维模型作为控制网格，而网格变形方法则使用自动创建的三维网格作为控制网格。这两种方法都是通过控制网格来改变三维模型的形状，但是外包框的控制网格位于三维模型的外部。而网格变形的方所采用的控制网格是一个规则的具有内部顶点的三维空间网格，这种网格可以和三维模型相交。

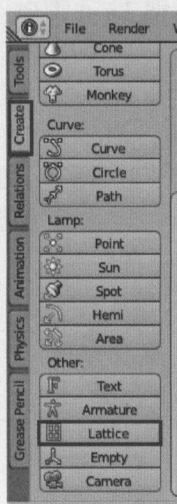


图 1-14 创建 Lattice 网格

网格变形方法主要通过生成一个 Lattice 控制网格，然后改变这个控制网格的形状，从而改变网格所包含的三维模型的形状。

第一步：导入要变形的兔子（bunny）三维模型。

第二步：创建一个控制网格（Lattice），如图 1-14 所示。

第三步：通过修改界面上网格（Lattice）的属性，可以确定网格的大小、位置，还可以改变它默认的行数、列数、层数。相关的界面和得到的网格如图 1-15 所示。

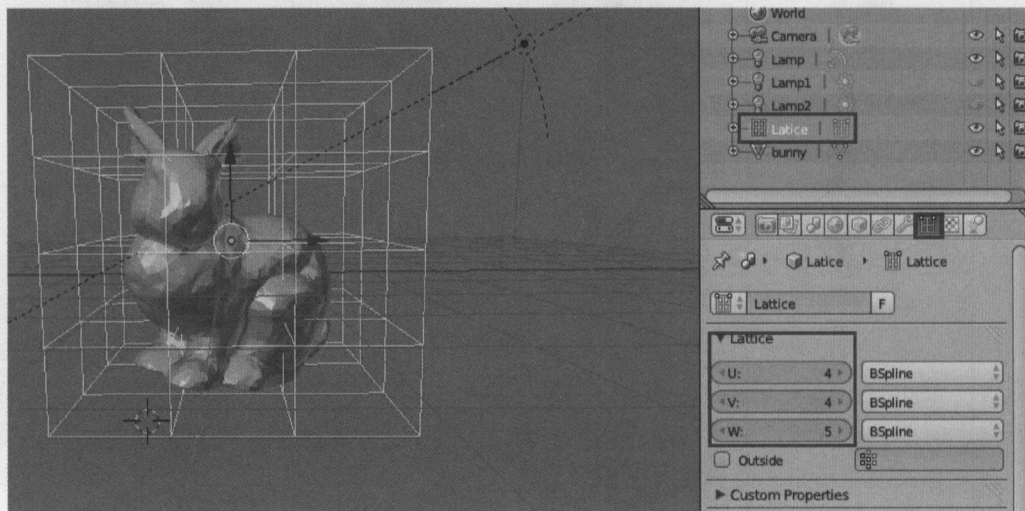


图 1-15 修改 Lattice 网格的属性

第四步：和外包框变形相似，需要为要变形的兔子模型添加一个修改器（Modifier），在修改器菜单里面选择 Lattice 这个选项，如图 1-16 所示。

第五步：如图 1-17 所示，Object 下拉列表具有 Lattice 属性的对象，选择这个 Lattice，就可以为三维模型绑定刚才生成的 Lattice 网格作为控制网格。

第六步：单击 Lattice 选择它作为控制网格，然后进入控制网格 Lattice 的编辑模式。

第七步：通过对 Lattice 上各个顶点的移动可以改变 Lattice 的形状，从而可以改变 Lattice 所包含的三维兔子模型的形状。操作界面和结果如图 1-18 所示。

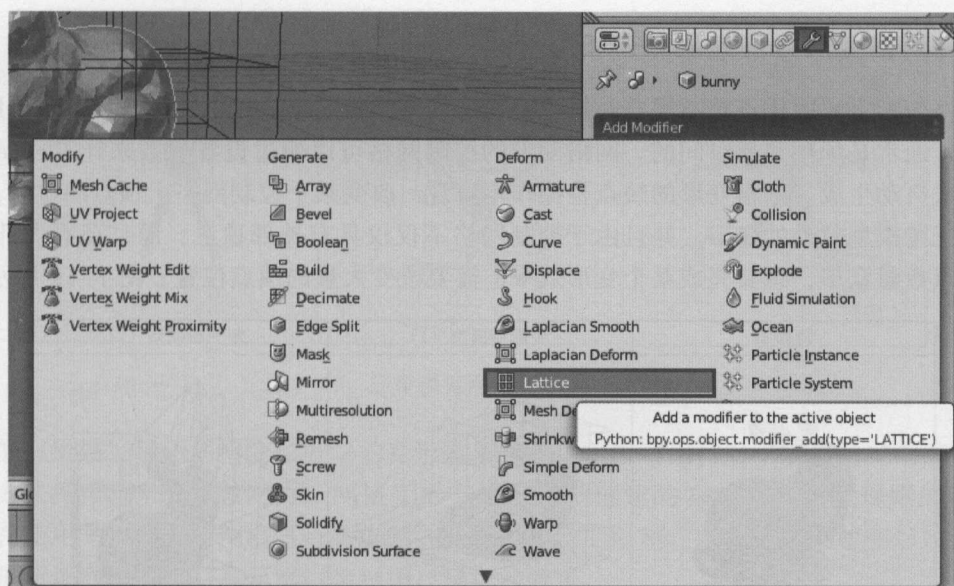


图 1-16 添加 Lattice 的 Modifier

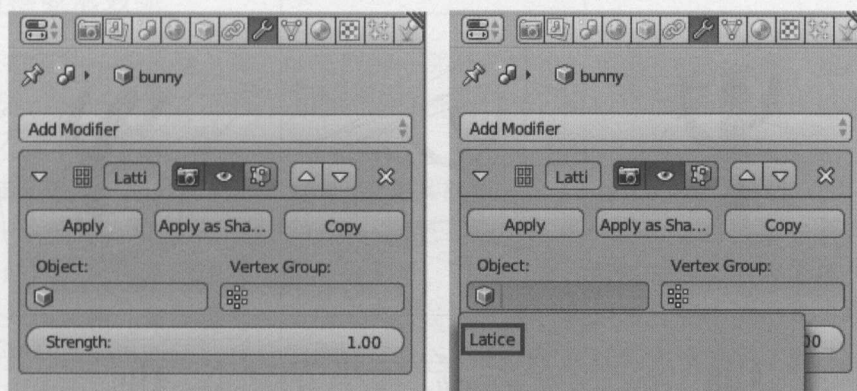


图 1-17 为模型绑定 Lattice 网格

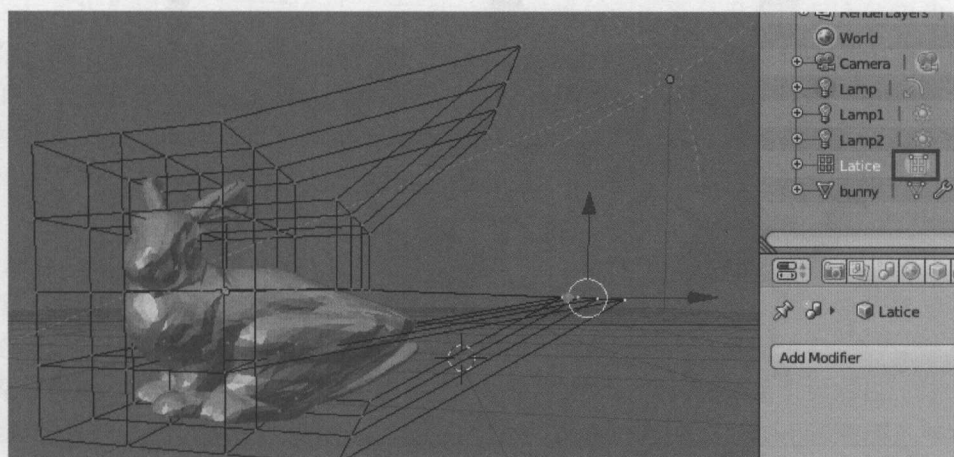


图 1-18 使用 Lattice 方法对模型进行变形

1.3.2 网格变形方法效果和分析

网格变形方法的优点是不需要对每个要变形的三维模型生成一个外包框。这个生成外包框的过程通常是烦琐和耗时间的。网格变形的控制网格可以通过设置参数来让 Blender 软件通过算法自动生成。网格变形的缺点是操作不直观，改变某个控制网格上顶点的位置，通常会引起三维模型意外的变形。并且由于控制网格不仅仅具有外部顶点，而且还有内部顶点，因此顶点数量众多，想要完成某个变形效果，需要改变大量的顶点位置。图 1-19 展示了各

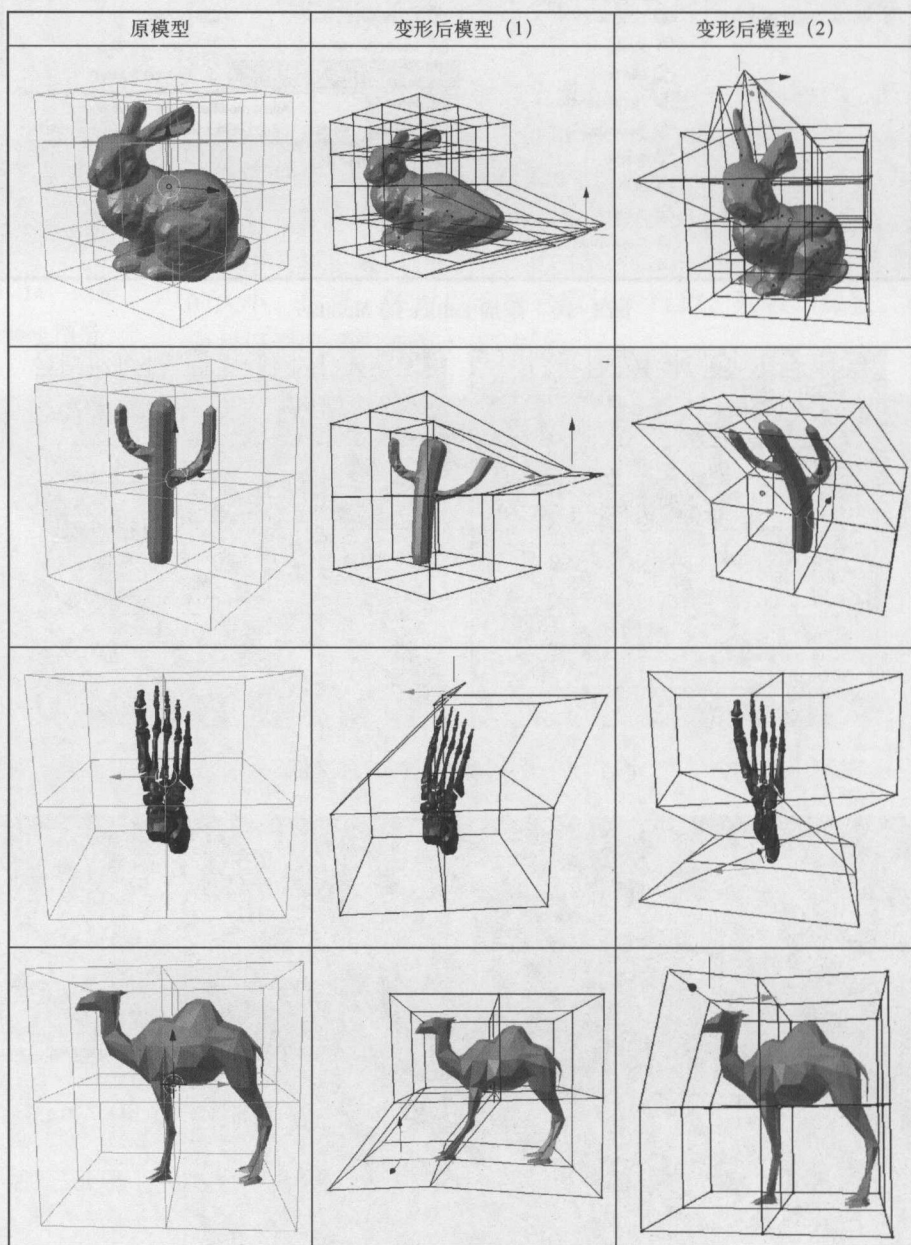


图 1-19 三维模型网格变形实验

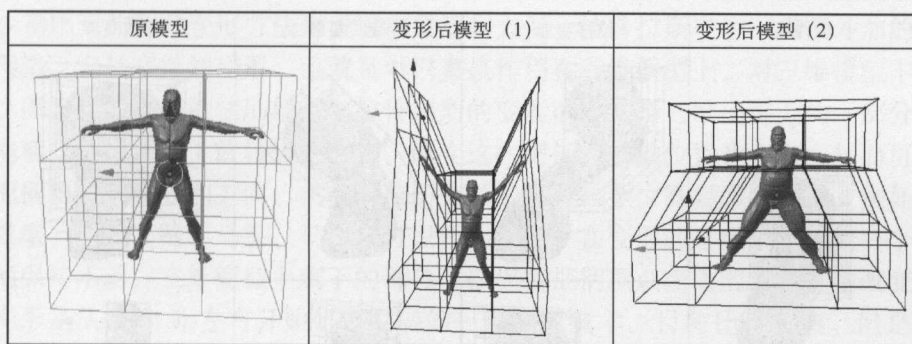


图 1-19 三维模型网格变形实验 (续)

种不同的三维模型采用网格变形方法的变形效果。从中可以看出,对于复杂的三维模型,网格变形方法操作起来就比较吃力。网格变形方法里自动创建的 Lattice 网格是规则的长方体网格,在使用其作为控制网格进行变形时,会受到一定的限制,想要达到某些变形效果时可能需要进行烦琐的操作,甚至难以达到用户预期的效果。

1.3.3 网格变形效果

本小节在各种三维模型上对三维模型的一部分和整个三维模型进行位移、旋转、缩放等变形,通过这些不同设置、不同方法的效果分析和对比,可以更加深入了解网格变形方法的特点。

1. 只对模型的一部分用网格 Lattice 变形实验

如图 1-20 所示,位移变形得到的第一幅图和第二幅图是用晶格的一部分点控制模型进行位移变形的图,可以从图中看出变形很平滑,没有出现扭曲等现象,效果不错,位移变形的第三幅图是用晶格的全部点来控制模型进行位移变形,可以看到整个模型都发生了位移变形,而不是只有晶格里面的部分发生位移变化,下面加了个挂钩也没能把下面晶格外的部分勾挂住;旋转变形的第一幅图和第二幅图是用晶格的部分点来控制模型旋转的变形,从图中可以看出变形平滑,效果很好,旋转变形的第三幅图是用晶格的所有点控制模型头部进行旋转变形的,可以看到整个模型都发生了旋转,而且晶格外面的部分发生了很大的扭曲;缩放变形的第一幅图是放大变形,缩放变形的第二幅图是缩小变形,这两幅图都是由晶格的部分点控制的,可以看出缩放变形都能实现平滑过渡,效果很好,缩放变形的第三幅图是晶格的所有点控制的缩小变形,可以看到整个模型都缩小了,只不过头部缩小得多,晶格外面的部分缩小得少,而通常希望的效果是晶格外面的部分不要缩小,但是在交接的地方能够平滑过渡。

2. 对整个模型使用网格 Lattice 变形修改器实验

如图 1-21 所示,从位移变形的两幅图中看出变形很平滑,但是模型的形状略微有些变化,但总体效果还是不错的;从旋转变形的两幅图中可以看出变形平滑,效果很好;缩放变形的第一幅图是放大变形,缩放变形的第二幅图是缩小变形,由这两幅图可以看出,缩放变形都能实现平滑过渡,效果很好,缺点是要想对细小的部位进行缩放可能比较麻烦,控制不是太方便。

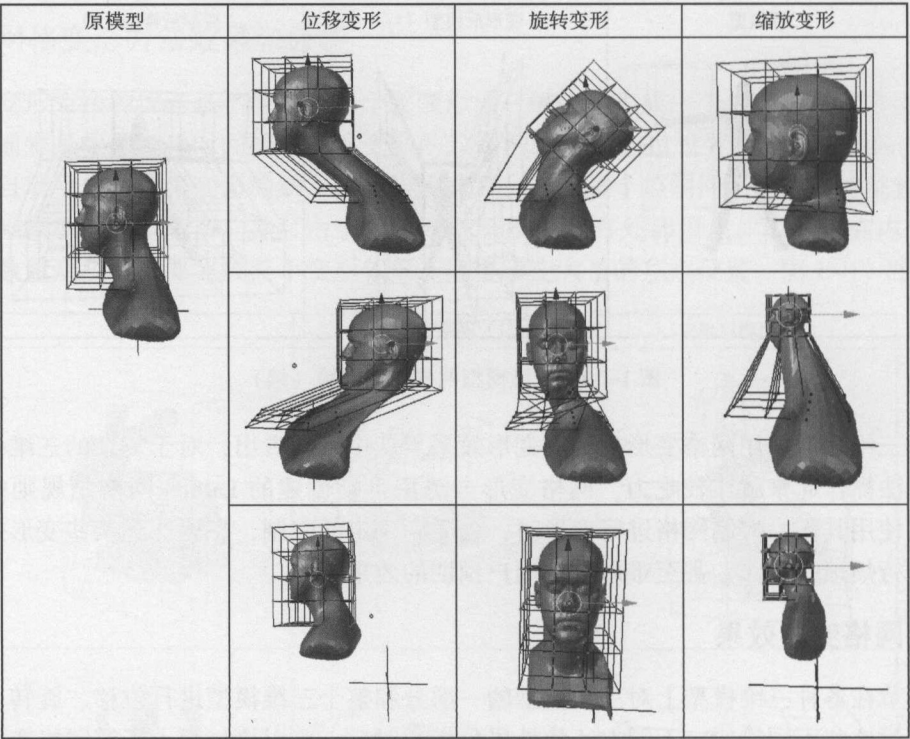


图 1-20 三维模型局部网格变形

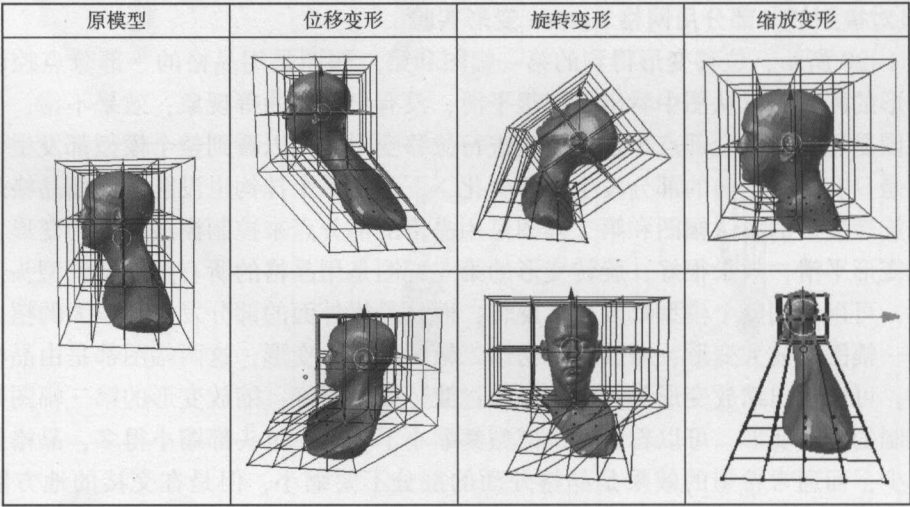


图 1-21 三维模型整体网格变形



1.4 拉普拉斯变形

1.4.1 拉普拉斯变形步骤

拉普拉斯变形（Laplacian Deform）是 Blender 实现的一种技术比较新的变形方法，这种

变形方法相比于前两种变形方法来说更加灵活和方便。拉普拉斯变形不需要外加的外包框或者控制网格，它是直接的变形。也就是可以直接作用在三维模型上，对三维模型本身直接进行变形，而不需要额外的辅助模型。拉普拉斯的变形需要选择三维模型上的一部分顶点作为变形的把柄，通过位移，旋转这些把柄，改变这些把柄顶点的位置和方向，从而可以改变其他不是把柄顶点的位置和方向。在拉普拉斯变形的时候，至少需要两个把柄。例如，一个马的三维模型，四个马蹄、马头、马尾巴可以作为把柄。在变形的过程中，改变一个把柄，其余的把柄保持不动，三维模型上剩下的部分可以根据把柄顶点当前的位置自动改变形状。拉普拉斯变形是从2004年左右开始研究发展的一种变形技术，目前比较成熟。但是大多数商业软件如3dsmax等还没有采纳。

拉普拉斯变形步骤主要是选择相应顶点作为变形把柄，移动把柄，从而得到变形结果。

第一步：导入要变形的三维模型。

第二步：按 Tab 键进入编辑模式，选中三维模型上一组相邻的点，按 Control + H 键选择“钩挂到一个新物体”选项，如图 1-22 所示。例如，选择三维模型上马蹄部分的点，如图 1-22 中马蹄上的黄色点所示，把这部分黄色的点钩挂到一个 Blender 提供的“十字形”物体上。这个 Blender 提供的“十字形”物体可以旋转、位移，从而带动所钩挂的一组顶点旋转、位移。

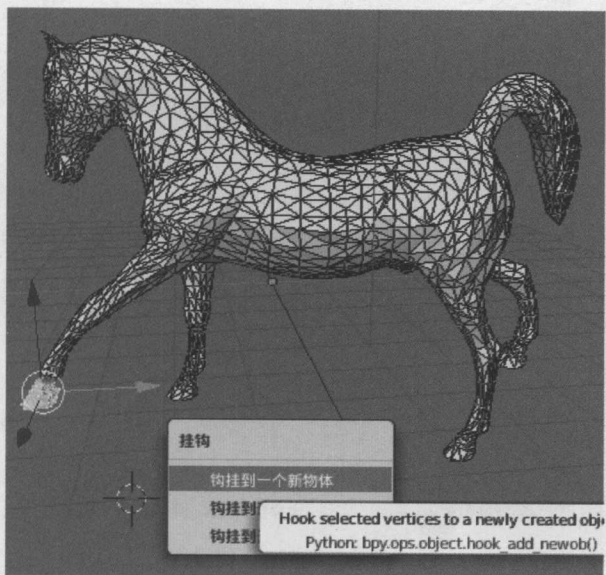


图 1-22 钩挂

第三步：按 Control + G 键，选择“指定到新组”选项，如图 1-23 所示。

第四步：对三维模型的其他三个马蹄、马头、马尾等部分重复第二步和第三步。

在图 1-24 中，所有黄色的点都分别钩挂到了六个“十字形”的物体上，并指定到同一个新组。

第五步：在三维模型修改器界面上，对三维模型添加一个拉普拉斯变形修改器 (Laplacian Deform)，如图 1-25 所示。

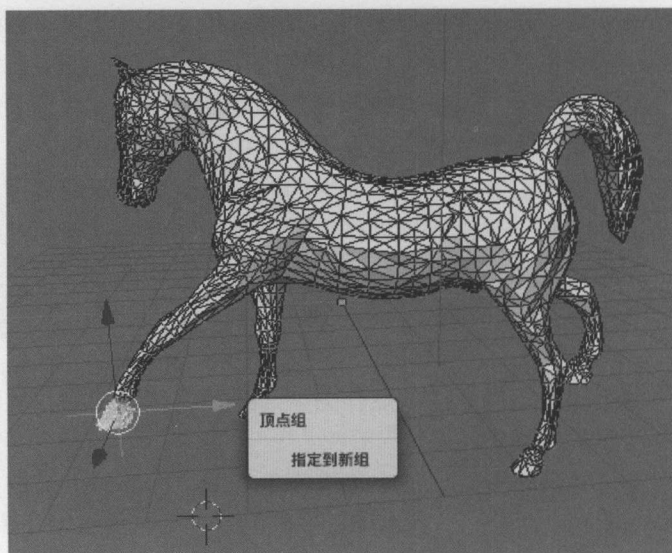


图 1-23 指定新组

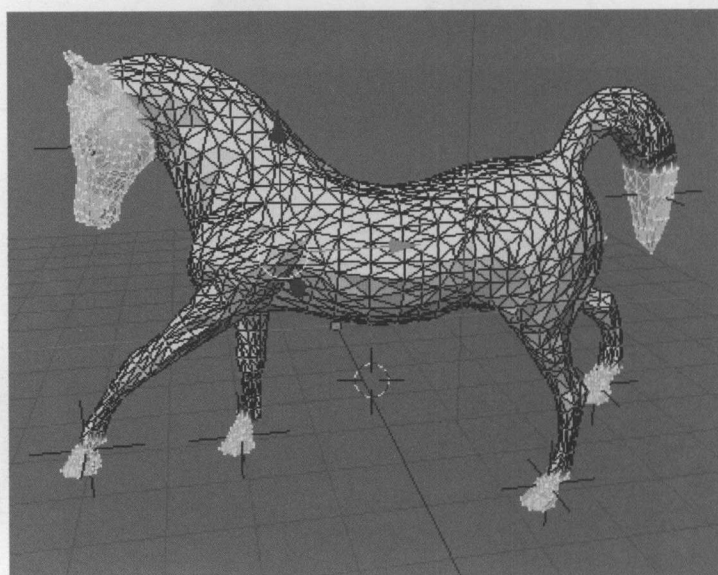


图 1-24 所有把柄的设置

第六步：在图 1-26（a）中选择群组，也就是刚才所有选择顶点指定的新组，最后绑定，这个操作把拉普拉斯变形修改器和三维模型上选择的点进行了绑定。图 1-26（b）是绑定后的界面，可以进行解绑。假如需要更换不同的选择点，则需要先解绑，然后再选择其他的顶点进行绑定。

第七步：在完成三维模型的编辑模式后返回到物体模式。选中其中一个“十字形”物体，按 G 键，移动鼠标就可对“十字形”进行旋转和位移，从而带动该“十字形”物体钩挂的顶点发生相应的旋转和位移，然后拉普拉斯变形器可以根据新的顶点位置改变三维模型的形状，如图 1-27 所示。

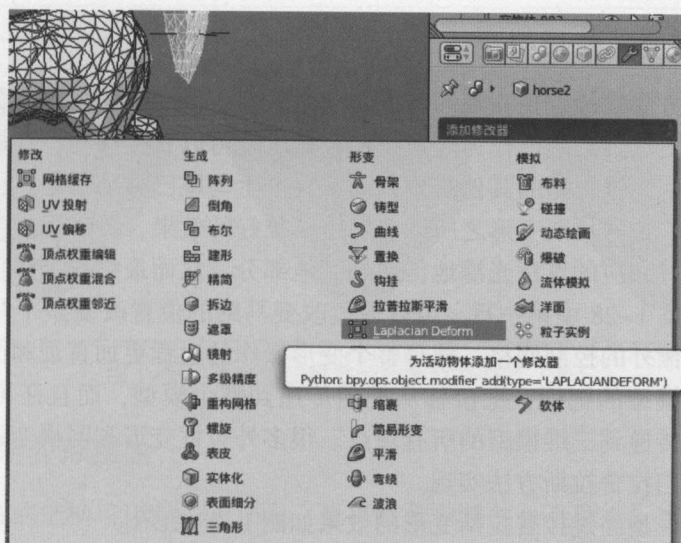


图 1-25 拉普拉斯修改器界面

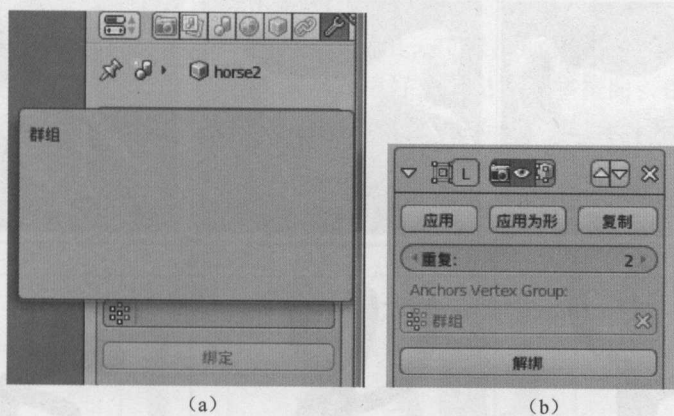


图 1-26 绑定界面

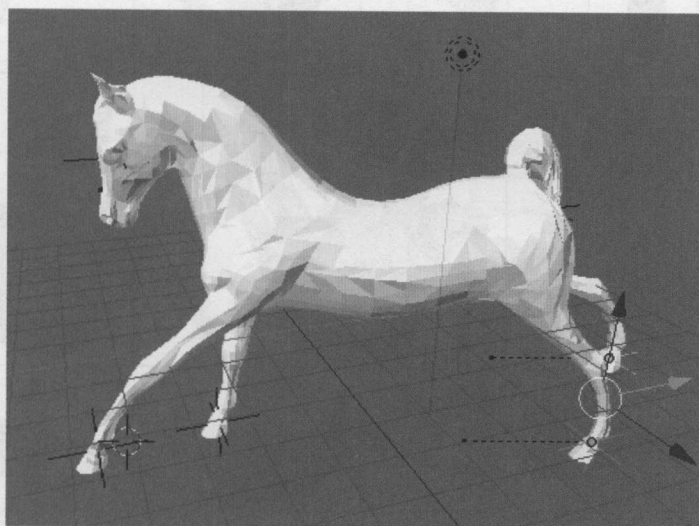


图 1-27 物体模式

1.4.2 拉普拉斯变形效果和分析

拉普拉斯变形方法的优点是直观，可以实现所见即所得的变形。例如，把马尾巴移动到另一个位置，只需要选择马尾巴作为变形把柄，然后就可以直接移动，马尾巴把柄产生的变形会被光滑地传递到三维模型的其他部位，而不会产生马尾巴移动但是其他部分不随着马尾巴部分移动的效果。拉普拉斯变形之所以能够得到很好的效果，关键思想就是直接移动的变形把柄部分可以通过相应的算法光滑地传递到其他部分，从而最终得到的仍然是一个光滑的三维模型。例如，图 1-28 中第一行，可以通过改变马蹄的位置改变整个马的姿势。并且拉普拉斯变形不需要额外的控制网格，使得整个变形操作的过程更加直观和简单。拉普拉斯变形不仅仅可以移动模型的把柄，把位移从把柄传递到整个模型，而且还可以旋转模型的把柄，把把柄的旋转传递到三维模型的所有顶点。很多外包框变形和网格变形方法难以实现的高难度变形都可以用拉普拉斯方法实现。

各种不同三维模型经过拉普拉斯变形的效果如图 1-28 所示。

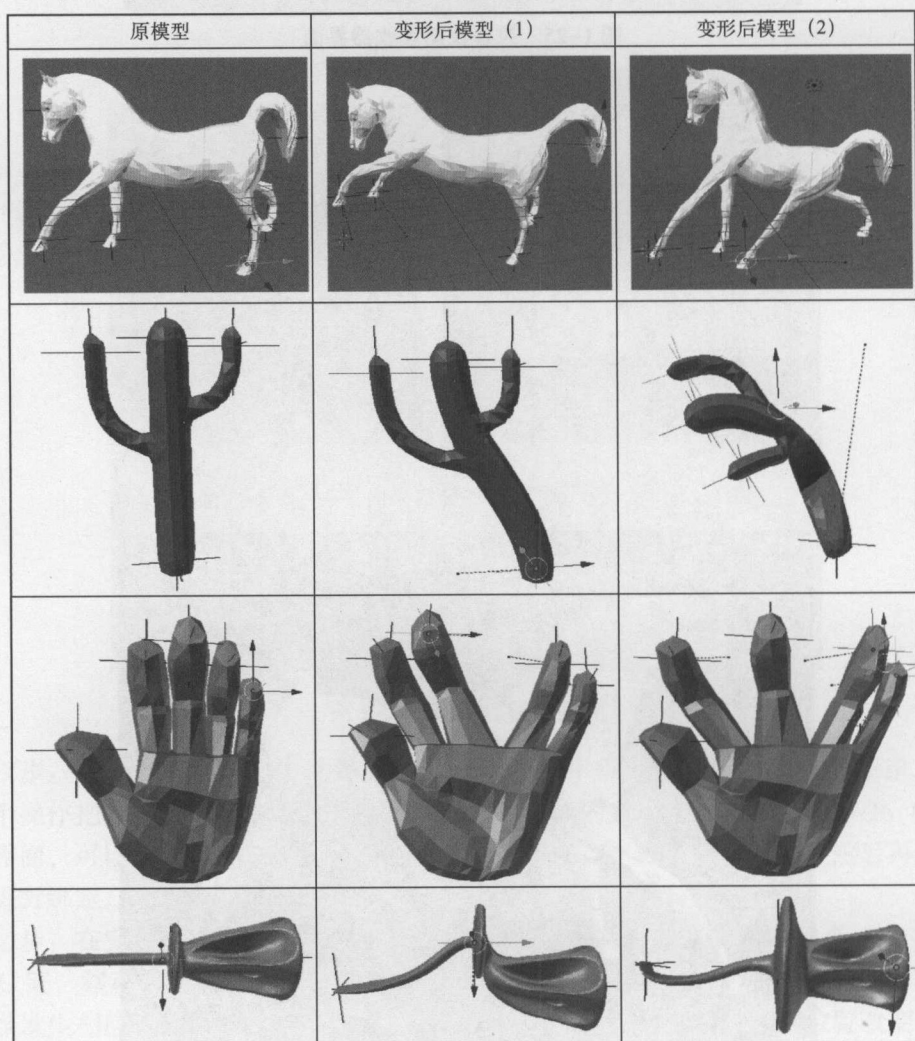


图 1-28 各种不同模型的拉普拉斯变形

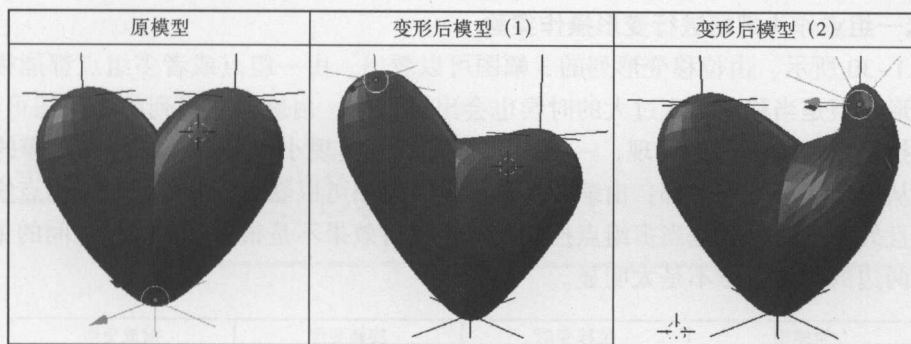


图 1-28 各种不同模型的拉普拉斯变形 (续)

1.4.3 拉普拉斯变形实验

拉普拉斯变形的把柄可以是一组顶点，也可以是单独的一个顶点，本小节通过一组顶点和单独的顶点分别进行位移、旋转、缩放的变形实验，展示拉普拉斯变形方法的效果。

1. 选一个点作为把柄进行变形实验

如图 1-29 所示，由位移变形的两幅图可以看出，拉普拉斯变形可以用一个点进行小幅度的位移变形，但是当位移幅度过大时效果就不是很好，会出现一些扭曲；由 3 个点控制的旋转变形也是可以实现小幅度变形的，当旋转到较大幅度，如旋转变形的图中所示的 180° 时就会产生扭曲，而且中间的主干没能平滑地过渡，当然只选中一个点旋转是不能实现的；拉普拉斯变形无法用一个点实现缩放变形，但是 2 个以上的点就能实现缩放，这和旋转有些类似。

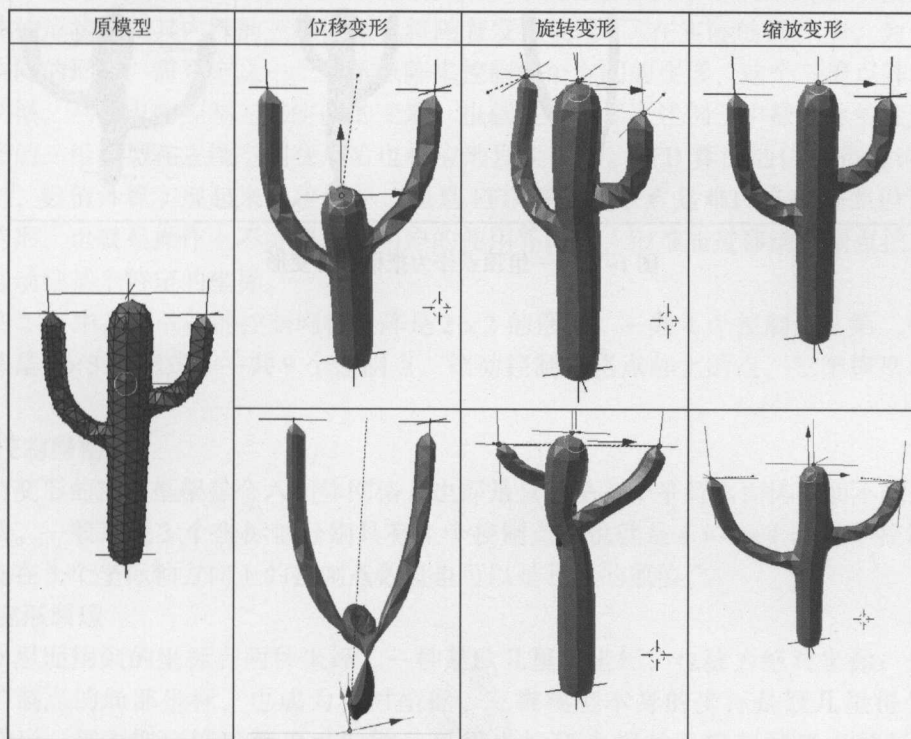


图 1-29 同一个模型的拉普拉斯变形实验

2. 选一组点作为把柄进行变形操作实验

如图 1-30 所示，由位移变形列的 3 幅图可以看出，由一组点或者多组点都能很好地实现位移变形，但是当移动幅度过大的时候也会出现扭曲；由旋转变形列的 3 幅图可以看出，由多组点控制旋转能很好地实现，一组点控制时可以实现小幅度旋转，旋转传递的范围有限，不能从模型底部传到顶部；由缩放变形列的 3 幅图可以看出，能够通过一组点实现缩放变形，而且效果很好，但是当多组点控制缩放变形时效果不是很好，仙人掌中间的主干变形明显，而两边的分支变形不是太明显。

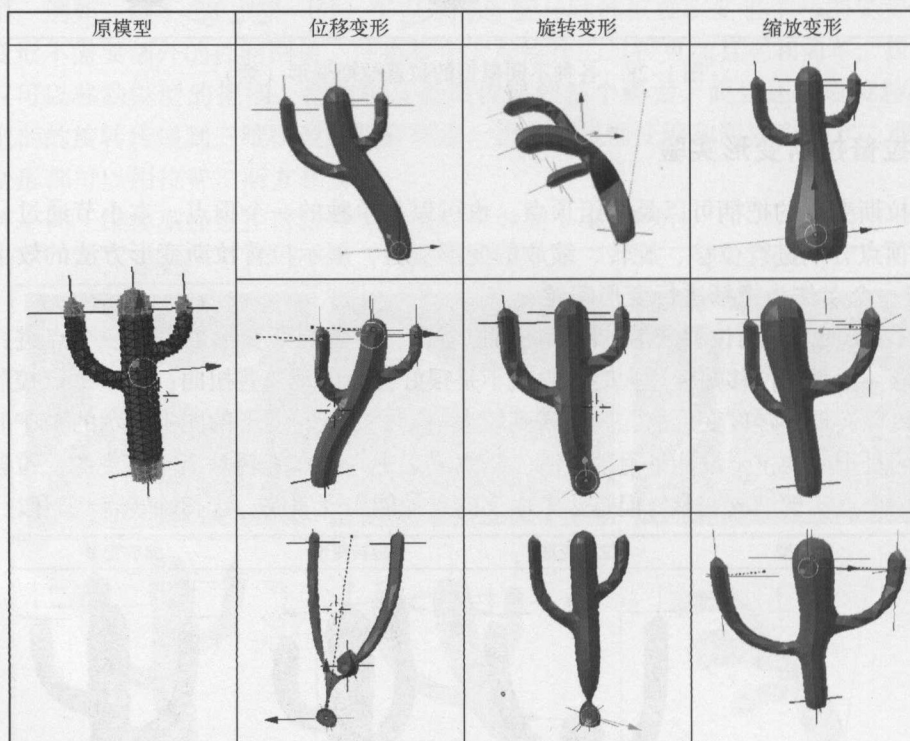


图 1-30 一组顶点作为把柄进行变形

第2章

FFD 变形算法



2.1 FFD 介绍

第1章主要讲述三种变形方法在软件里的应用和操作。通过 Blender 软件提供的界面,可以采用3种不同的方法来对三维模型进行变形。本章和后续几章分别讲述在 Blender 软件里实现这3种变形方法底层所依赖的原理和计算机算法,以及这些算法的实现过程。和 Blender 软件里面的网格变形方法相对应的变形算法称为 FFD 变形算法,本章主要讲述 FFD 变形算法的原理和实现。

FFD (Free - Form Deformation) 变形也称为自由变形或网格变形,这个变形方法不直接作用于三维模型,而是间接作用于三维模型。FFD 变形方法首先对三维模型所在三维空间进行变形,然后三维空间变形后,带动里面的三维模型发生相应的变形。自由变形的过程可以做这样的物理类比:“假设有一个橡皮体,其内部镶嵌着所需要变形的三维模型。若改变橡皮体的形状,则其内部的三维模型也将随着发生变形”。在实际的操作中,为了方便改变三维空间的形状,需要定义一个三维点阵来控制整个空间的变形,这个三维点阵称为变形的控制网格。三维点阵控制三维空间的变形,也就是相对于上述例子中橡皮体的变形,想要进行变形的三维模型在三维空间变形后也相应地发生变形。FFD 算法的优点是理论上直观,容易理解,数值计算实现起来比较简单。但是 FFD 算法的缺点是难以准确按照设计者的意图完成变形,也就是操作上不方便。在用户的使用界面上,很难通过移动控制点把三维模型上的点移动到某个特定的坐标。

在图 2-1 中,第一行的控制网格点阵是 2×2 的形式,一共 4 个控制点;第二行的控制网格点阵是 3×3 的形式,一共 9 个控制点。拉动控制网格点阵上的点,三维模型就会发生变化。

1. 控制网格

自由变形的控制框架是个六面体网格,也即是只能是一个平行六面体,而不能是任意拓扑的形状。一般来说 3 个坐标轴分别具有 4 个控制点,也就是 $4 \times 4 \times 4 = 64$ 个控制点的网格,但是在 3 个坐标轴方向上的控制点数量也可以是其他的数值。

2. 变形原理

FFD 里面用到的坐标有两种坐标,一种是欧几里得坐标,也称为绝对坐标;另一种是相对于控制点的局部坐标,也成为相对坐标。三维模型本身的坐标是欧几里得坐标,为了进行变形,需要把三维模型顶点的欧几里得坐标变为相对于控制网格点的局部坐标。局部坐标表明了三维模型顶点相对于控制网格顶点的相对位置。在变形的过程中,需要

保持这个相对位置，也就是局部坐标不发生改变。从而控制网格点的位置变动后，在保持三维模型上每个顶点相对于控制网格顶点的局部坐标不变的前提下，根据当前的控制网格顶点的绝对坐标和三维模型顶点的局部坐标就可以计算出三维模型上顶点的绝对坐标，也就是得到了三维模型顶点变形后的坐标，从而使三维模型发生了变形。因此，整个变形过程可以分为下面的步骤。

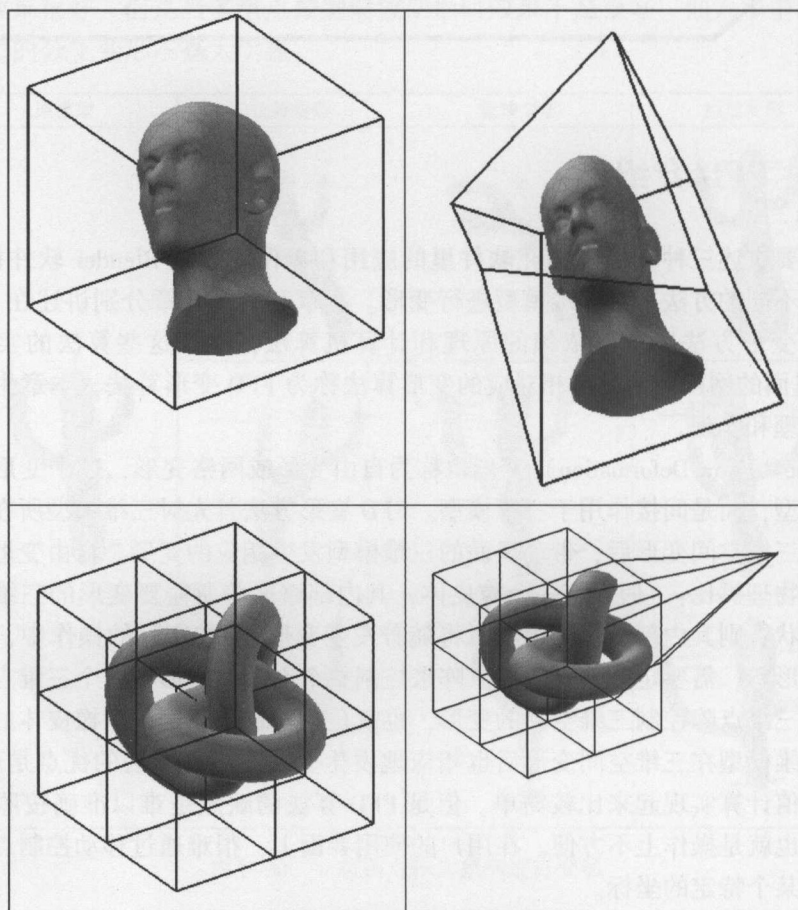


图 2-1 FFD 变形效果图

- (1) 确定要变形的空间；
- (2) 在此空间上构造一个长方体网格，此网格由控制点构成；
- (3) 把三维模型放到变形空间里；
- (4) 计算三维模型和控制网格上控制点的相对坐标；
- (5) 移动控制点，对控制网格进行变形；
- (6) 计算三维模型的新位置。



2.2 FFD 算法数学推导

FFD 变形的核心数学计算是从三维模型顶点的欧几里得坐标计算局部坐标，以及从局部

坐标和控制点的新位置计算三维模型顶点的绝对坐标。数学计算方法的推导过程如下。

第一步：在三维空间定义一个局部坐标系，该坐标系由一个原点 X_0 和 3 个两两垂直的向量 S 、 T 、 U 组成，构成了一个平行六面体的区域，如图 2-2 所示。

第二步：图 2-2 中的三维模型就是想要进行变形的模型。对于这个三维空间里的任意点 X ，都可以定义一个局部坐标 (s, t, u) ，这个局部坐标和 X 的关系为：

$$X = X_0 + sS + tT + uU$$

第三步：点 X 的 (s, t, u) 局部坐标通过线性代数可以得出为：

$$s = \frac{T \times U \cdot (X - X_0)}{T \times U \cdot S}, \quad t = \frac{S \times U \cdot (X - X_0)}{S \times U \cdot T}, \quad u = \frac{S \times T \cdot (X - X_0)}{S \times T \cdot U}$$

第四步：对于三维空间里任何一点的局部坐标 (s, t, u) 都满足如下条件：

$$0 < s < 1, \quad 0 < t < 1, \quad 0 < u < 1$$

第五步：在上述三维空间中添加一个 $(l+1) \times (m+1) \times (n+1)$ 的平行六面体控制网格，将要变形的三维空间包含在其中，如 2-3 图定义了 $3 \times 3 \times 3$ 和 $4 \times 4 \times 4$ 两种不同的控制网格。

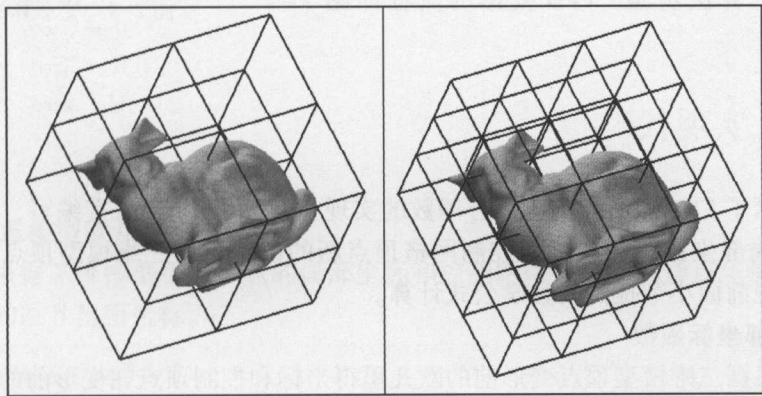


图 2-3 三维空间里的控制网格

第六步：控制网格上的各点可以表示为：

$$X_0 + \frac{i}{l}S + \frac{j}{m}T + \frac{k}{n}U = P_{ijk}$$

第七步：其中 $i = \{0, 1, \dots, l\}$ ， $j = \{0, 1, \dots, m\}$ ， $k = \{0, 1, \dots, n\}$ ，

P_{ijk} 表示的是控制网格上某一点的欧几里得坐标。

第八步：在控制网格上的点的位置改变之后，也就是平行六面体空间进行被变形后，这个三维空间内每一点的欧几里得坐标可以通过下面公式求得：

$$X_{ffd} = \sum_{i=0}^l \binom{l}{i} (1-s)^{l-i} s^i \left[\sum_{j=0}^m \binom{m}{j} (1-t)^{m-j} t^j \left[\sum_{k=0}^n \binom{n}{k} (1-u)^{n-k} u^k P_{ijk} \right] \right]$$



2.3 FFD 算法步骤

在定义了欧几里得坐标和局部坐标的相互转换之后，就可以实现 FFD 的算法。整个算法过程可以分为初始化和更新两个大的模块。

1. 初始化 (Initialize) 模块

第一步：在载入变形模型之后生成控制网格。

第二步：设置局部坐标的坐标原点及三个坐标轴向量。

第三步：根据公式计算变形模型上各点的 (s, t, u) 坐标，创建数组进行记录。

2. 更新 (update) 模块

当控制网格上的点被移动时进行如下操作。

第一步：找到变形模型上一点，按照公式对点的位置重新进行计算，并进行更新。

第二步：重复第一步，直至变形模型上所有点的位置都完成更新。

根据上述的算法步骤，FFD 变形的流程如图 2-4 所示。

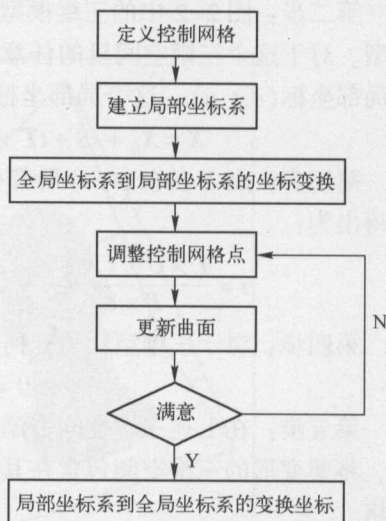


图 2-4 变形算法流程图



2.4 实现代码

本小节展示了 FFD 算法代码里核心函数的实现，代码采用 C# 语言编写，主要分为计算 FFD 算法里的局部坐标函数和根据控制网格顶点新的位置更新三维模型顶点的绝对坐标函数，也就是实现前面小节提到的数学公式计算。

1. 计算局部坐标函数

这个函数根据三维模型顶点变形前的欧几里得坐标和控制顶点在变形前的相对位置来计算三维模型顶点的局部坐标，也就是相对坐标。这个相对坐标在变形的过程中，保持不变。

```

public void BuildCoordianteFFD()
{
    FFDs = new List<double>();
    FFDt = new List<double>();
    FFDu = new List<double>();
    boxi = new List<int>();
    boxj = new List<int>();
    boxk = new List<int>();
    Vector3D max = TriMeshUtil.ComputeBoundingBox(cage).Max;
    Vector3D min = TriMeshUtil.ComputeBoundingBox(cage).Min;
    Vector3D X0 = TriMeshUtil.ComputeBoundingBox(cage).Min;
}
    
```

```

Vector3D length = TriMeshUtil. ComputeBoundingBox( cage ). Max
                    - TriMeshUtil. ComputeBoundingBox( cage ). Min;
Vector3D S = new Vector3D( length. x, 0. 0, 0. 0 );
Vector3D T = new Vector3D( 0. 0, length. y, 0. 0 );
Vector3D U = new Vector3D( 0. 0, 0. 0, length. z );
foreach ( TriMesh. Vertex vertex in mesh. Vertices )
{
    double s = T. Cross( U ). Dot( vertex. Traits. Position - X0 ) / T. Cross( U ). Dot( S );
    double t = U. Cross( S ). Dot( vertex. Traits. Position - X0 ) / U. Cross( S ). Dot( T );
    double u = S. Cross( T ). Dot( vertex. Traits. Position - X0 ) / S. Cross( T ). Dot( U );
    FFDs. Add( s );
    FFDt. Add( t );
    FFDu. Add( u );
}
foreach ( TriMesh. Vertex vertex in box. Vertices )
{
    int i = ( int ) ( ( vertex. Traits. Position - X0 ). x * ( CageConfig. Instance. S - 1 ) / S. x );
    int j = ( int ) ( ( vertex. Traits. Position - X0 ). y * ( CageConfig. Instance. T - 1 ) / T. y );
    int k = ( int ) ( ( vertex. Traits. Position - X0 ). z * ( CageConfig. Instance. U - 1 ) / U. z );
    boxi. Add( i );
    boxj. Add( j );
    boxk. Add( k );
}
}

```

2. 更新三维模型欧几里得坐标

这个函数根据三维模型每个顶点的局部坐标和控制网格变形后的顶点坐标来计算三维模型每个顶点新的欧几里得坐标。

```

public void UpdateFFD()
{
    foreach ( TriMesh. Vertex vertex in mesh. Vertices )
    {
        Vector3D position = new Vector3D( 0. 0, 0. 0, 0. 0 );
        foreach ( TriMesh. Vertex bvertex in box. Vertices )
        {
            position += sigFunction( ( CageConfig. Instance. S - 1 ),
                                    ( boxi[ bvertex. Index ] ),
                                    ( FFDs[ vertex. Index ] ) ) * sigFunction( ( CageConfig. Instance. T - 1 ),
                                    ( boxj[ bvertex. Index ] ),
                                    ( FFDt[ vertex. Index ] ) ) * sigFunction( ( CageConfig. Instance. U - 1 ),
                                    ( boxk[ bvertex. Index ] ),
                                    ( FFDu[ vertex. Index ] ) ) * bvertex. Traits. Position;
        }
    }
}

```

```

    }
    vertex.Traits.Position = position;
}
}

```

3. 辅助函数

```

private int factorial(double i)
{
    int a = 1;
    if (i > 0)
    {
        for (int j = (int)i; j > 0; j --)
        {
            a *= j;
        }
        return a;
    }
    else
    {
        return a;
    }
}

private double sigFunction(int l, int i, double s)
{
    return factorial(l)/factorial(i)/factorial(l-i)
        * Math.Pow(1-s, l-i) * Math.Pow(s, i);
}

```


第3章

均值坐标变形算法



3.1 均值坐标介绍

本章的均值坐标变形和下一章要讲的格林坐标变形算法都是外包框变形算法，和 Blender 软件里外包框变形的方法相对应，都是基于在三维模型外部构建一个外包框来对三维模型进行变形。

均值坐标变形算法是使用均值坐标对三维模型进行变形的一种方法。这种方法和 FFD 的区别是，外部的控制是一个把三维模型包围的框，而不是一个控制网格。控制网格上的点是个矩阵形状的点，而均值坐标用的包围框上的点只出现在三维模型外部，而不出现在三维模型的内部。区别如图 3-1 所示。

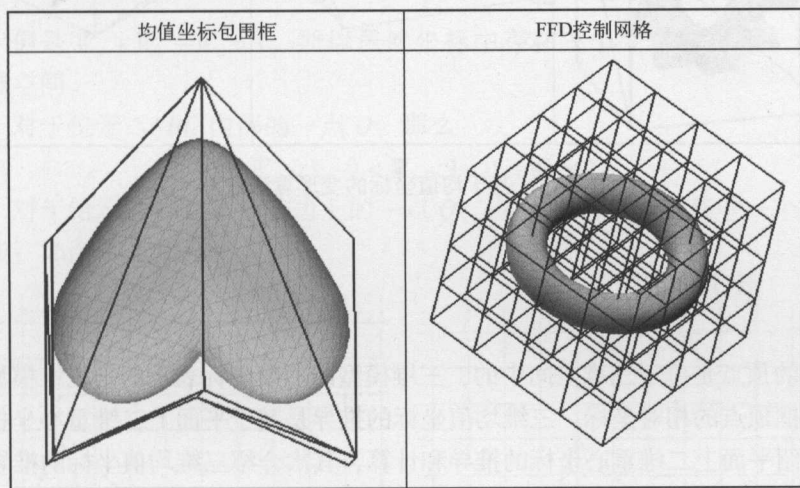


图 3-1 包围框和控制网格对比

基于均值坐标变形的核心思想是把三维模型上的每个点表示为外部包围框上点的相对坐标，这个相对坐标在变形过程中是不变的。当外部包围框发生变化，也就是包围框的顶点位置改变后，再根据包围框上顶点的新位置和之前计算出来的相对坐标来计算三维模型上顶点的新位置。

整个变形方法分为三步。

第一步：针对想要变形的三维模型上的每一个顶点，都逐一确定控制包围框上各点的权重，也就是相对坐标，或者均值坐标。

第二步：在变形的过程中，保持均值坐标不变，对控制外包框进行操作，改变外包框的形状，也就是控制外包框上的顶点位置发生了改变，从而这些顶点的世界坐标也发生了变化。

第三步：在根据控制外包框变化后的绝对坐标和要变形模型上各个顶点的均值坐标进行相应计算，就可以唯一地确定变化后三维模型上各点新的绝对坐标，从而就使三维模型形状发生了变化。

均值坐标变形的思路很清晰，关键点在于如何计算这个权重。对于平面的三角形来说，权重很容易计算，可以通过重心坐标来实现。计算均值坐标的思路是把重心坐标从三角形扩展到多边形，然后再从二维平面的多边形扩展到三维空间的多面体。给定一个三维空间多面体，把三维空间的点表示为此多面体上各个顶点的权重，这一步是各种基于包围框进行变形算法的核心。根据计算方法不同有各种权重，如均值坐标、格林坐标、和谐坐标等。

图 3-2 中的均值变形采用立方体作为外包框，立方体是最简单的外包框，改变立方体顶点的位置，就可以使立方体包含的三维模型形状发生改变。

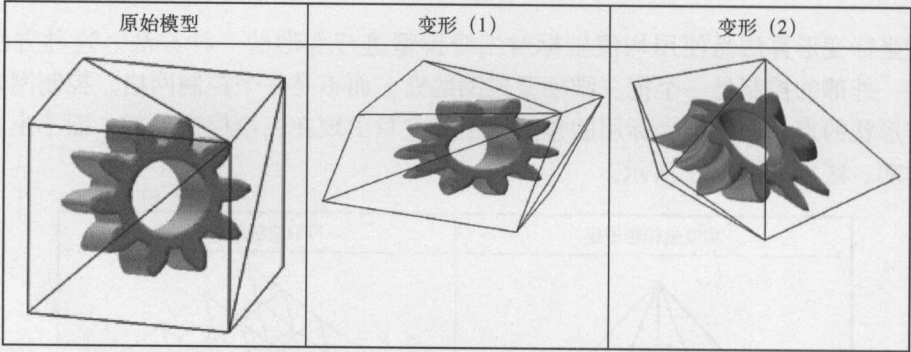


图 3-2 基于均值坐标的变形算法效果



3.2 重心坐标

三维模型的顶点是位于三维空间中的，三维模型的均值坐标表示的是三维模型顶点相对于三维模型外包框顶点的相对坐标。三维均值坐标的推导是基于平面上二维重心坐标的推导。这一节，首先介绍平面上二维重心坐标的推导和计算，其次介绍三维均值坐标的推导和计算。

1. 概述

在一个平面当中，给定一个任意的三角形，平面上的任意一点都可以表示为三角形三个顶点的加权平均值，而这个的权重，就叫作该点相对于三角形三个顶点的重心坐标，即重心坐标是一个相对坐标，是相对于三角形三个顶点的。

2. 计算方法

第一步：将平面上一点分别与平面上给定的一个三角形的三个顶点连接。

第二步：当该点不在三角形的任意一边上时，该点会分别与给定三角形的三条边构成三个新的三角形，当点在给定三角形的某一条边上时，则分别与其他两条边构成两个新的三角形。

第三步：这个点的重心坐标就是新构成的三角形面积和原三角形面积的比值，即对于平面上的一点，给定三角形的各点权重就等于该点对边与平面上一点所构成的三角形面积比上给定三角形的面积。

第四步：当平面上一点处于给定三角形之外时，在构成的3个新三角形中，假如和给定的三角形完全不存在重叠的部分，那么这个三角形对应顶点的权值为负数。

3. 重心坐标计算示例

第一步：在图3-3中，给定 $\triangle ABC$ ，3个顶点在平面中的坐标分别为：

$$P_a(X_a, Y_a), P_b(X_b, Y_b), P_c(X_c, Y_c)$$

第二步：对任意一点 O ，它的坐标为 $P_o(X_o, Y_o)$ 。

第三步：将 O 与三角形三个顶点分别相连，即连接 OA 、 OB 、 OC 。

第四步：计算 W_a 、 W_b 、 W_c 的值：

$$W_a = S_{\triangle BOC} / S_{\triangle ABC}, W_b = S_{\triangle AOC} / S_{\triangle ABC}, W_c = S_{\triangle AOB} / S_{\triangle ABC}$$

第五步： O 在该平面中以 $\triangle ABC$ 为基准的重心坐标表示为 $M_o(W_a, W_b, W_c)$ ，其中：

$$P_o = W_a \cdot P_a + W_b \cdot P_b + W_c \cdot P_c$$

第六步：在图3-3中计算得到：

$$S_{\triangle ABC} = 44, S_{\triangle AOB} = 17, S_{\triangle BOC} = 11, S_{\triangle AOC} = 16$$

第七步：因此

$$W_a = \frac{1}{4}, W_b = \frac{4}{11}, W_c = \frac{17}{44}$$

第八步：但是 $W_a + W_b + W_c = 1$ ，所以虽然坐标中存在三个变量，实际上只有两个自由度，仍是二维空间。

第九步：对于位于 $\triangle ABC$ 内部的一点 O ，那么：

$$0 < W_a < 1, 0 < W_b < 1, 0 < W_c < 1$$

第十步：对于给定 $\triangle ABC$ 某一条边上的一点 O ，点 O 的重心坐标其中一个元的值为0。

第十一步：如图3-4所示，由于：

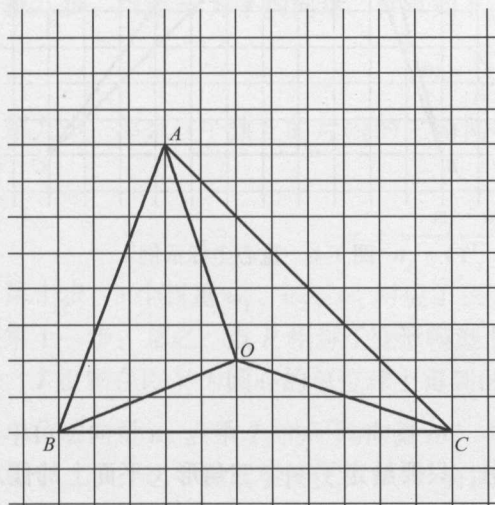


图3-3 重心坐标示例1

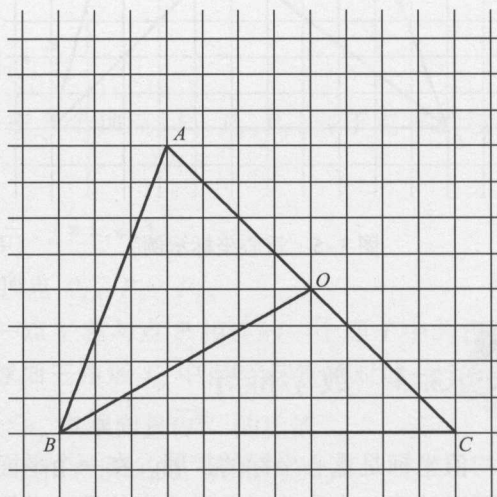


图3-4 重心坐标示例2

$$S_{\triangle ABC} = 44, S_{\triangle AOB} = 22, S_{\triangle BOC} = 22, S_{\triangle AOC} = 0$$

因此计算出来的重心坐标为：

$$W_a = \frac{1}{2}, W_b = 0, W_c = \frac{1}{2}$$

第十二步：对于给定 $\triangle ABC$ 外的一点 O ，点 O 的重心坐标中有至少一个、至多两个的值为负数。

第十三步：如图 3-5 所示，因为 $\triangle AOC$ 与给定 $\triangle ABC$ 完全不存在重叠部分：

$$S_{\triangle ABC} = 44, S_{\triangle AOB} = 25.5, S_{\triangle BOC} = 38.5, S_{\triangle AOC} = 20$$

所以：

$$W_a = \frac{7}{8}, W_b = -\frac{5}{11}, W_c = \frac{51}{88}$$

但是仍然满足 $W_a + W_b + W_c = 1$ 。

第十五步：而对图 3-6 的示例，点 O 的重心坐标中有两个元的值为负数。其中， $\triangle AOB$ 、 $\triangle AOC$ 与给定 $\triangle ABC$ 完全不存在重叠部分：

$$S_{\triangle ABC} = 44, S_{\triangle AOB} = 4.5, S_{\triangle BOC} = 60.5, S_{\triangle AOC} = 12$$

从而

$$W_a = \frac{11}{8}, W_b = -\frac{3}{11}, W_c = -\frac{9}{88}$$

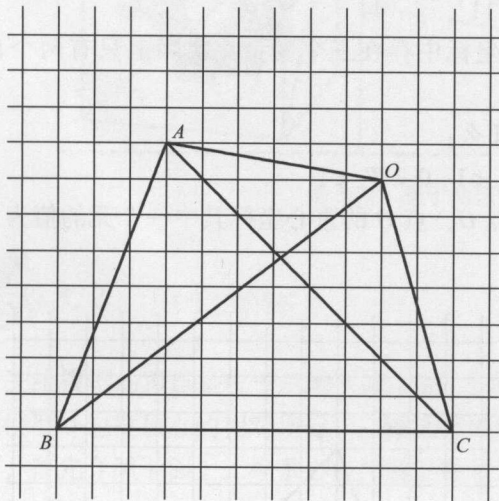


图 3-5 重心坐标示例 3

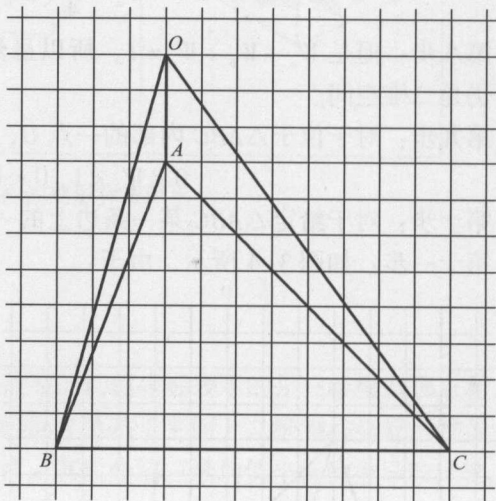


图 3-6 重心坐标示例 4



3.3 数学推导

均值坐标是重心坐标的扩展。在一个平面中，只要给定了一个三角形，平面上的任意一点都可以计算一个和此三角形对应的重心坐标，从而可以通过这个三角形把原来的欧几里得坐标变换为重心坐标。重心坐标是基于三角形的，均值坐标就是重心坐标在多边形上的扩

展。也就是给定一个多边形,把平面上的点表示为基于这个多边形上各个顶点的权重。在三维的情况下,就是给定一个多面体,把三维空间的任意一点表示为这个多面体上各个顶点的权重,或者相对坐标。

也就是对于三维空间的一个有 n 个顶点的多面体

$$P = \{p_1, p_2, \dots, p_n\}$$

那么位于此空间中的任意一点 V , 都可以确定一组以给定多面体各个顶点为基准的权重

$$W = \{w_1, w_2, \dots, w_n\}$$

而点 V 的位置可以通过给定多面体上各点位置的加权平均值来表示, 即

$$v = \frac{\sum_i w_i p_i}{\sum_i w_i}$$

均值坐标推导过程如下。

第一步: 包围框上的一个点用 P 来表示, 三维空间或者三维模型上的点用 V 来表示。

第二步: 把包围框上的 3 个点构成的一个面指定为 T , 如图 3-7 所示。

第三步: 对于这个三角形面 T , 3 个顶点为 $\{P_1, P_2, P_3\}$, 将其投影到以 V 为中心的单位球 S_v 上, 得到球面三角形 T_p 。

第四步: 这个球面三角形和点 V 构成一个球面三棱锥, 这个三棱锥除了球面外, 还有 3 个面, 这 3 个面的法向为 n_1, n_2, n_3 。

第五步: 这 3 个法向 $\{n_1, n_2, n_3\}$ 和 3 个顶点 $\{P_1, P_2, P_3\}$ 一一对应。

第六步: n_j 是由点 V 和球面三角形 T_p 的第 j 条边确定的平面的向内单位法线向量。

第七步: 指定球面三角形 T_p 第 j 条边的边长为 L_j 。

第八步: 指定 m 为平均向量, 计算如下:

$$m = \sum_{j=1}^3 L_j \cdot \frac{n_j}{2}$$

第九步: 那么这个独立的片段的三角网格中 3 个顶点 $\{P_1, P_2, P_3\}$ 对于点 V 的权重分别为

$$w_j = \frac{n_j \cdot m}{n_j \cdot (P_j - V)}, j=1, 2, 3$$

第十步: 3 个权重 w_1, w_2, w_3 对应了三个顶点 P_1, P_2, P_3 。

第十一步: 总之, 点 P 相对于变形模型上一点 V 在与点 P 相连的一个面 T 中的权重 w_T 等于“ T 投影在以 V 为圆心的单位球上得到的球面三角形 T_p 中, 与点 P 相对的一个侧面的向内单位法向量 n_p 点乘 T_p 的平均向量 m ”与“ n_p 点乘向量 \vec{VP} ”的比值。

第十二步: 这个权重不仅仅和 V, P 有关, 还和 P 所在的三角形有关。

第十三步: 上面计算的权重 w_T 只表示了控制网格上一点 P 对于模型上一点 V 在 P 的某个面上的权重。而实际上与 P 相连的除了 T 还有其他三角形面。为了计算点 P 相对于 V 的

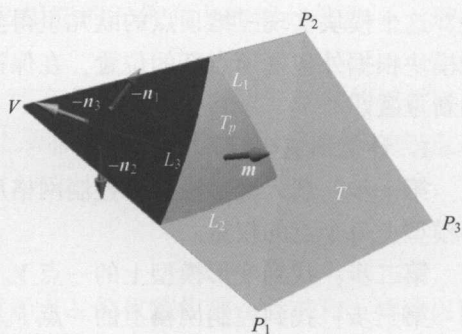


图 3-7 均值坐标构建图示

权重，就需要遍历控制网格上所有与点 P 相连的面 $T_s = \{T, T', T'', \dots\}$ ，进行相同的计算，再对所有的 w_T 求和，即

$$w_P = \sum_{T_n \in T_s} \frac{n_{PT_n} \cdot m_{T_n}}{(n_{PT_n} \cdot (P - V))}$$

第十四步：根据上述方法可以计算三维空间一点对于外包框上所有点的权重。

第十五步：遍历三维模型，计算三维模型上每一点对于外包框的权重，也就是均值坐标。



3.4 变形步骤

通过均值坐标进行三维模型的变形，也分为两大模块。第一个模块是均值坐标计算模块，这个模块实现三维顶点的欧几里得坐标到均值坐标的变换；第二个模块是更新模块，这个模块根据外包框顶点新的位置，在保持均值坐标不变的前提下，计算三维模型上每个顶点的新位置。

1. 计算均值坐标

第一步：载入变形模型和控制网格后，创建二维数组，用以记录控制网格上各点针对变形模型上每个点的权重。

第二步：找到变形模型上的一点 V 。

第三步：找到控制网格上的一点 P 。

第四步：找到与点 P 相连的一个三角面上的三个点（其中一点为 P ），与模型上的点 V 构建球面三棱锥，求出三个棱的单位向量。

```
Vector3D a = ComputeTetDir( vertex, p1 );
Vector3D b = ComputeTetDir( vertex, p2 );
Vector3D c = ComputeTetDir( vertex, p3 );

private Vector3D ComputeTetDir( TriMesh. Vertex vertex, TriMesh. Vertex p1 )
{
    Vector3D a1 = ( p1. Traits. Position - vertex. Traits. Position )
        / Math. Sqrt( Math. Pow( ( p1. Traits. Position. x - vertex. Traits. Position. x ), 2 )
        + Math. Pow( ( p1. Traits. Position. y - vertex. Traits. Position. y ), 2 )
        + Math. Pow( ( p1. Traits. Position. z - vertex. Traits. Position. z ), 2 ) );
    return a1;
}
```

第五步：通过 3 个单位向量的两两叉乘，求出球面三棱锥三个侧面的法向量。

```
n1 = a2. Cross( a3 ) * ( -1 );
n2 = a3. Cross( a1 ) * ( -1 );
n3 = a1. Cross( a2 ) * ( -1 );
```

第六步：并且通过 3 个单位向量求反三角函数，得到球面三角形的三条边长。


```

L1 = Math. Acos( a2. Dot( a3 ) );
L2 = Math. Acos( a3. Dot( a1 ) );
L3 = Math. Acos( a1. Dot( a2 ) );

```

第七步：根据公式，利用球面三角形的三条边长及球面三棱锥的三个侧面的法向量求得平均向量 m ；

```

m = ( L1 * n1 + L2 * n2 + L3 * n3 ) / 2;

```

第八步：根据公式，求得与点 P 相关的该三角面上的 w_j ；

```

w = n1. Dot( m ) / n1. Dot ( cage. Vertices[ i ]. Traits. Position
    - vertex. Traits. Position );

```

第九步：重复第四步至第七步，求得所有与点 P 相连的三角面上的部分权重 w_j ，进行求和，得到点 P 对于模型上点 V 的权重。

第十步：重复第三步至第八步，得到控制网格上所有点对于变形模型上点 V 的权重。

第十一步：重复第二步至第九步，得到控制网格上所有点对于变形模型上每一点的权重。

2. 变形部分

第一步：移动外包框的顶点。

第二步：找到变形模型上一点 V 。

第三步：将控制网格上各点变化后的位置与其对于点 V 的权重相乘，并求和，再除以所有对于点 V 的权重之和，得到点 V 的新位置，进行更新。

第四步：重复进行第二步和第三步，直至变形模型上所有点的位置都完成更新。

3. 均值坐标变形核心函数

均值坐标核心代码分为两个主要的函数：计算均值坐标和更新三维模型位置。

1) 计算均值坐标函数

```

public List < double[ ] > BuildCoordianteMeanValue( )
{
    Vector3D a1, a2, a3, n1, n2, n3, m;
    double L1, L2, L3, w;
    List < double[ ] > meanValue = new List < double[ ] > ( );
    foreach ( TriMesh. Vertex vertex in mesh. Vertices )
    {
        meanValue. Add( new double[ cage. Vertices. Count ] );
        for ( int i = 0; i < cage. Vertices. Count; i ++ )
        {
            meanValue[ vertex. Index ][ i ] = 0;
            foreach ( TriMesh. Face face in cage. Vertices[ i ]. Faces )
            {
                //得到三角面上的三个点
                TriMesh. Vertex p1 = face. GetHalfedge( 0 ). FromVertex;
                TriMesh. Vertex p2 = face. GetHalfedge( 1 ). FromVertex;

```

```

        TriMesh.Vertex p3 = face.GetHalfedge(2).FromVertex;
        //求出球面三棱锥三个棱的单位向量
        Vector3D a = ComputeTetDir(vertex, p1);
        Vector3D b = ComputeTetDir(vertex, p2);
        Vector3D c = ComputeTetDir(vertex, p3);
        if (p1.Index == cage.Vertices[i].Index)
        {
            a1 = a;
            a2 = b;
            a3 = c;
        }
        else if (p2.Index == cage.Vertices[i].Index)
        {
            a3 = a;
            a1 = b;
            a2 = c;
        }
        else
        {
            a2 = a;
            a3 = b;
            a1 = c;
        }
        //得到三个侧面的法向量

        n1 = a2.Cross(a3) * (-1);
        n2 = a3.Cross(a1) * (-1);
        n3 = a1.Cross(a2) * (-1);
        //得到球面三角形的三个边长
        L1 = Math.Acos(a2.Dot(a3));
        L2 = Math.Acos(a3.Dot(a1));
        L3 = Math.Acos(a1.Dot(a2));
        //求出 m
        m = (L1 * n1 + L2 * n2 + L3 * n3) / 2;
        w = n1.Dot(m) / n1.Dot(cage.Vertices[i].Traits.Position
            - vertex.Traits.Position);
        meanValue[vertex.Index][i] += w;
    }
}

return meanValue;
}

```

2) 更新三维模型函数

```

public void UpdateMeanValue()
{
    foreach (TriMesh.Vertex vertex in mesh.Vertices)
    {
        if (meanValue == null)
        {
            throw new Exception("Mean value coordinate is not built");
        }
        else
        {
            Vector3D position = new Vector3D(0.0,0.0,0.0);
            double amountw = 0;
            for (int i=0; i < cage.Vertices.Count; i++)
            {
                position += meanValue[vertex.Index][i] * cage.Vertices[i].Traits.Position;
                amountw += meanValue[vertex.Index][i];
            }
            vertex.Traits.Position = position/amountw;
        }
    }
}

```



3.5 效果分析

均值坐标主要原理是将原网格模型顶点表示成为控制外包框顶点的线性组合。均值坐标就是这个线性组合的权重系数。均值坐标是一种齐次坐标，具有非负性、仿射不变性、拉格朗日插值、光滑性、内部局部性、非退化性等性质。可以将封闭多边形或网格中的任意点表示为其边界顶点的线性组合。使用均值坐标的网格变形，具有计算效率高，能够保持光滑性等特征。

均值坐标变形的缺点对控制外包框的形状要求比较高。首先，在变形模型非凸的部分，可能会产生不自然扭曲，这是由均值坐标公式的特性决定的。其次，三维模型上的点会受到距离比较近的外包框的点影响。这个距离是欧几里得直线距离，而不是沿着三维模型表面的测地距离，从而移动外包框上点的时候，和这个点临近的三维模型上的点都会受到影响。在试图移动模型的某一部分时，可能会发生期望之外的形变。例如，在对人物模型进行变形时，人物模型的两腿分别向两侧张开时，两腿的粗细都发生了变化，而实际上并不是变形时期望模型发生这样的形变。在如图3-8所示的变形中，也表现出了均值坐标的这一缺点。例如，图3-9右边图中的斧柄发生了错位，而在图3-10中的模型发生了严重的扭曲，基本已经难以保持原来的结构。

1. 实验一

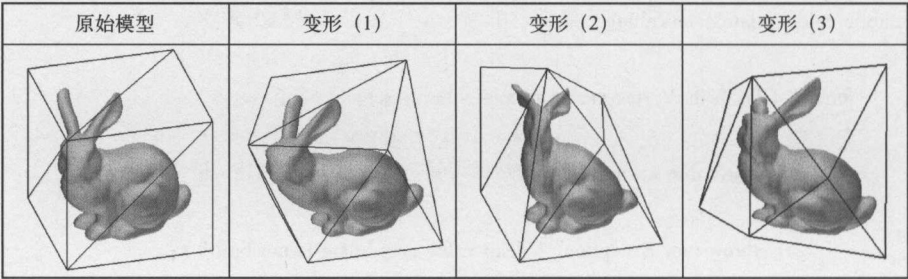


图 3-8 兔子变形实验效果

2. 实验二

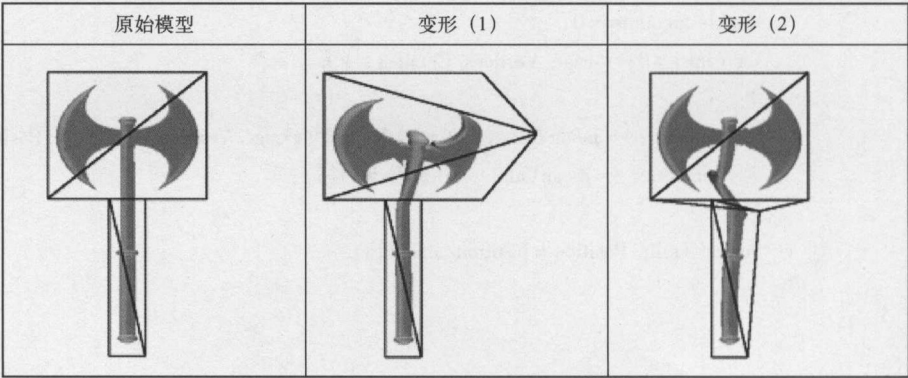


图 3-9 斧头变形实验效果

3. 实验三

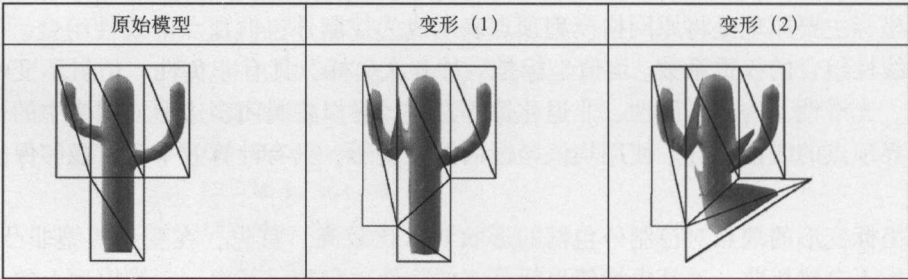


图 3-10 仙人掌变形实验效果

第 4 章

格林坐标变形算法



4.1 格林变形介绍

格林坐标变形和均值坐标变形一样，也是基于外包框的变形方法，但是具有一些均值坐标不具有的优点。在均值坐标变形算法中，每个三维模型顶点的均值坐标只有一组值，而和均值坐标不一样，每个三维模型顶点的格林坐标有两组值。格林坐标虽然也不满足内部局部性，但能很好地保持非负性。格林公式的这一特性，使得在变形过程中，避免了某些操作者预期之外的扭曲。通过一定的拓展，格林坐标能够产生在包围盒的边界仍然是连续的映射，因此在外包框只包含三维模型的一部分的情况下，还可以使用格林坐标来改变三维模型。格林坐标变形算法可以使被包含部分三维模型发生的改变能够平滑地过渡到没有被外包框所包含的三维模型区域。格林坐标具有仿射不变性，但在模型变形的过程中，可能会包含剪切及各项异性的缩放，因此导致变形过程中不能保持模型本身的某些特征。在格林坐标变形过程中，每个坐标轴方向的变换都是独立的，与其他轴方向的变换互不干扰。因此，有时会无法很好地表现出某些控制网格具有的特征，即也可能无法准确地按照操作者预期的效果进行变形。

图 4-1 是格林坐标变形的实例，图中的仙人掌三维模型通过外包框和格林坐标可以做各种不同的变形。

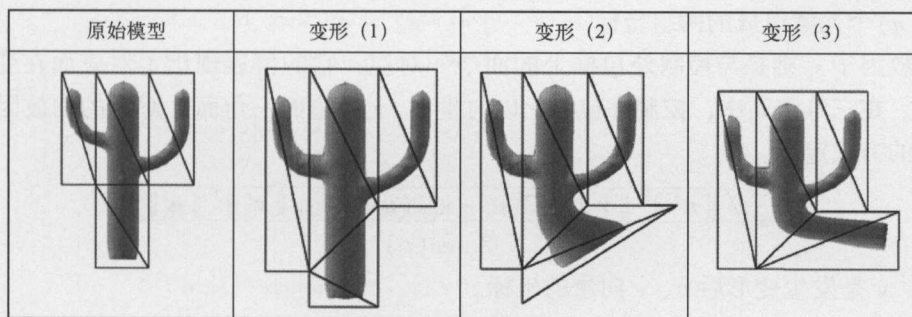


图 4-1 基于格林坐标的仙人掌模型变形

和均值坐标相比，格林坐标能够得到更光滑的变形，如在图 4-2 的实验中，和上一节的均值变形相比，斧子模型在把柄处没有发生错位。

和均值坐标变形算法一样，格林坐标变形算法也需要把三维模型内部顶点的欧几里得坐标转换为相对于外包框顶点的局部坐标，也称为相对坐标。但是在格林坐标中，相对坐标是

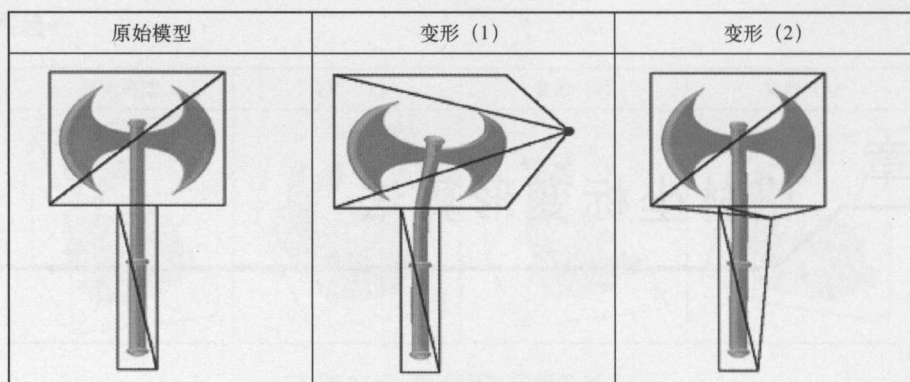


图 4-2 基于格林坐标的斧子模型变形

两组数值。格林坐标不仅仅利用了控制外包框上各点的位置信息，还使用了控制外包框上各个面的法向量。因此在某些情况下格林坐标变形算法可以更好地使变形模型表现出控制外包框期待的某些特征。

在格林坐标计算中，外包框内部的点和外包框上点的关系用下面公式来计算：

$$\eta = \sum_{i \in I_v} \phi_i v_i + \sum_{j \in J_T} \psi_j s_j n_j$$

其中， $\eta \in R^n$ 是控制外包框内部的任意点坐标，也就是要变形模型上任意一点的坐标；

$v_i \in R^n$ 是控制外包框上各个顶点的坐标；

$n_j \in R^n$ 是控制外包框上各个三角面的法向量；

$\{v_i\}_{i \in I_v} = V$ 和 $\{n_j\}_{j \in J_T} = T$ 分别是控制网格上顶点与三角面的集合；

$s_j \in R$ 是缩放因子，对于变形中的一些特性，如缩放不变至关重要；

ϕ_i 和 ψ_j 就是变形模型上 η 点的格林坐标。

格林坐标的两组数值分别表示为： ϕ_i 和 ψ_j 。这两组坐标分别和控制外包框上的点和面一一对应。也就是假如控制网格上有 m 个点和 n 个面，那么对于变形模型上任意一点，都有一组由 $(m+n)$ 个元素组成的两组坐标。

而缩放因子 s_i 则是与控制外包框上的面一一对应，它的值表现出了对应面在变形后的拉伸程度。在三维空间中，控制外包框上以向量 u 、 v 为边的三角面 t_j 对应的缩放因子 s_i 可以用下面的方式定义：

$$s_j = \frac{\sqrt{\|u'\|^2 \|v'\|^2 - 2(u' \cdot v')(u \cdot v) + \|v'\|^2 \|u\|^2}}{\sqrt{8} \text{area}(t_j)}$$

其中， u' 、 v' 是发生变形后 u 、 v 向量的坐标。



4.2 算法步骤和代码

整个格林坐标变形算法也分为两个模块，计算格林坐标模块和在保持格林坐标不变的前提下，根据外包框新的位置更新三维模型顶点位置模块。具体变形流程如下。

让 $\{v_i\}_{i \in I_v} = V$ 表示控制外包框上顶点的集合；格林坐标公式可以简化为：

$$\eta = \sum_{i \in I_V} \varphi_i(\eta) v_i$$

在控制外包框的顶点位置发生改变后,再通过:

$$\eta' = \sum_{i \in I_V} \varphi_i(\eta) v'_i$$

求出控制外包框内任意点的新坐标。

1. 计算格林坐标模块

第一步:载入变形模型和控制外包框后,创建二维数组用以记录格林坐标,创建一维数组用以记录缩放因子。

第二步:将所有数值初始化为0,同时创建一个列表用以记录控制网格上各个面的信息,将其值初始化为当前状态下控制网格上各个面的信息。

第三步:找到变形模型上的任意一点,遍历控制网格上的面,求解该点的格林坐标,代码如下。

```
public void BuildCoordianteGreen()
{
    Vector3D[] faceNormal = TriMeshUtil. ComputeNormalFace( cage );
    GreenCV = new List<double[]>();
    GreenCF = new List<double[]>();
    faceSj = new double[ cage. Faces. Count ];
    formFace = new TriMesh. Face[ cage. Faces. Count ];
    for ( int i = 0; i < cage. Faces. Count; i ++ )
    {
        formFace[ i ] = cage. Faces[ i ];
    }
    foreach ( TriMesh. Vertex vertex in mesh. Vertices )
    {
        GreenCV. Add( new double[ cage. Vertices. Count ] );
        GreenCF. Add( new double[ cage. Faces. Count ] );
        for ( int i = 0; i < cage. Vertices. Count; i ++ )
        {
            GreenCV[ vertex. Index ][ i ] = 0;
        }
        for ( int i = 0; i < cage. Faces. Count; i ++ )
        {
            GreenCF[ vertex. Index ][ i ] = 0;
        }
    }

    foreach ( TriMesh. Vertex vertex in mesh. Vertices )
    {
        for ( int i = 0; i < cage. Faces. Count; i ++ )
```

```

{
    TriMesh. Vertex p1 = cage. Faces[i]. GetHalfedge(0). FromVertex;
    TriMesh. Vertex p2 = cage. Faces[i]. GetHalfedge(1). FromVertex;
    TriMesh. Vertex p3 = cage. Faces[i]. GetHalfedge(2). FromVertex;

    Vector3D a1, a2, a3;
    a1 = p1. Traits. Position - vertex. Traits. Position;
    a2 = p2. Traits. Position - vertex. Traits. Position;
    a3 = p3. Traits. Position - vertex. Traits. Position;

    Vector3D n, p;
    n = faceNormal[i];
    p = a1. Dot(n) * n;

    double s1, s2, s3;
    s1 = Math. Sign(( (a1 - p). Cross(a2 - p)). Dot(n));
    s2 = Math. Sign(( (a2 - p). Cross(a3 - p)). Dot(n));
    s3 = Math. Sign(( (a3 - p). Cross(a1 - p)). Dot(n));

    Vector3D o = new Vector3D(0.0, 0.0, 0.0);
    double I1, I2, I3;
    I1 = GCTriInt(p, a1, a2, o);
    I2 = GCTriInt(p, a2, a3, o);
    I3 = GCTriInt(p, a3, a1, o);

    double II1, II2, II3;
    II1 = GCTriInt(o, a2, a1, o);
    II2 = GCTriInt(o, a3, a2, o);
    II3 = GCTriInt(o, a1, a3, o);

    Vector3D q1, q2, q3;
    q1 = a2. Cross(a1);
    q2 = a3. Cross(a2);
    q3 = a1. Cross(a3);

    Vector3D N1, N2, N3;
    N1 = q1 / (q1. Length());
    N2 = q2 / (q2. Length());
    N3 = q3 / (q3. Length());

    double g1 = s1 * I1 + s2 * I2 + s3 * I3;
    double g2 = Math. Abs(g1);
}

```

```

double I = g2 * (-1.0);
GreenCF[ vertex. Index ][ i ] = -1.0 * I;
Vector3D w = n * I + N1 * I1 + N2 * I2 + N3 * I3;

if ( w. Length() > 0.0001 )
{
    for ( int j = 0; j < cage. Vertices. Count; j ++ )
    {
        if ( cage. Vertices[ j ]. Index == p1. Index )
        {
            GreenCV[ vertex. Index ][ j ] += N2. Dot( w ) / N2. Dot( a1 );
        }
        if ( cage. Vertices[ j ]. Index == p2. Index )
        {
            GreenCV[ vertex. Index ][ j ] += N3. Dot( w ) / N3. Dot( a2 );
        }
        if ( cage. Vertices[ j ]. Index == p3. Index )
        {
            GreenCV[ vertex. Index ][ j ] += N1. Dot( w ) / N1. Dot( a3 );
        }
    }
}
}
}

```

第四步： v_{j_1} 、 v_{j_2} 、 v_{j_3} 是控制网格上任意一面的三个顶点， $\mathbf{n}(t_j)$ 是该面指向外侧的单位法向量，GCTriInt 是自定义的函数，内容如下。

```

private double GCTriInt( Vector3D p, Vector3D v1, Vector3D v2, Vector3D n )
{
    double a = Math. Acos( ( v2 - v1 ). Dot( p - v1 ) /
        ( v2 - v1 ). Length() / ( p - v1 ). Length() );
    double b = Math. Acos( ( v1 - p ). Dot( v2 - p ) /
        ( v1 - p ). Length() / ( v2 - p ). Length() );
    double c = Math. Pow( ( p - n ). Length(), 2 );
    double d = Math. Pow( ( p - v1 ). Length(), 2 )
        * Math. Pow( Math. Sin( a ), 2 );
    double S1, S2, C1, C2;
    S1 = Math. Sin( Math. PI - a );
    S2 = Math. Sin( Math. PI - a - b );
    C1 = Math. Cos( Math. PI - a );
}

```



```

C2 = Math. Cos( Math. PI - a - b);
double I1 = Math. Sign( S1)/( -2.0) *
        (2 * Math. Sqrt(c) * Math. Atan( C1 * Math. Sqrt(c)
        /Math. Sqrt( d + c * Math. Pow( S1,2) ) ) + Math. Sqrt( d)
        * Math. Log10(2 * Math. Sqrt( d) * Math. Pow( S1,2)
        /Math. Pow( (1 - C1),2)
        * (1 - 2 * c * C1/(c * (1 + C1) + d
        + Math. Sqrt( Math. Pow( d,2) + d * c * Math. Pow( S1,2) ) ) ) ) );
double I2 = Math. Sign( S2)/( -2.0) * (2 * Math. Sqrt(c)
        * Math. Atan( C2 * Math. Sqrt(c)
        /Math. Sqrt( d + c * Math. Pow( S2,2) ) ) + Math. Sqrt( d)
        * Math. Log10(2 * Math. Sqrt( d)
        * Math. Pow( S2,2)/Math. Pow( (1 - C2),2)
        * (1 - 2 * c * C2/(c * (1 + C2) + d
        + Math. Sqrt( Math. Pow( d,2) + d * c * Math. Pow( S2,2) ) ) ) ) );
double l = b * Math. Sqrt( c);
double j = Math. Abs( I1 - I2 - l);
double k = j * ( -1.0)/4.0/Math. PI;
return k;
    }
    
```

格林坐标中与控制外包框上各顶点对应的一部分坐标，计算方式与上一章讨论的均值坐标类似，在与该点相连的各个控制外包框的三角面上求得该值的一部分，然后进行求和。而格林坐标中与控制外包框上各个面对应的一部分坐标，则是与其对应的面指向外侧的单位法向量，以及由变形模型上一点指向该面的顶点的向量相关。

第五步：重复第三步，求得变形模型上所有点的格林坐标。

2. 更新三维模型顶点位置模块

第一步：根据列表记录变化前控制网格上各个面的信息，以及变化后（当前状态下）控制网格上各个面的信息，计算与各个面对应的缩放因子。

第二步：找到变形模型上的任意一点 V ，根据公式计算其新的位置，即首先将其格林坐标中与控制网格各点对应的部分坐标乘以其相对应的控制网格各点的新坐标，然后将其格林坐标中与控制网格各个面对应的部分坐标乘以其相对应的控制网格各个面新的指向外侧的单位法向量再乘以对应的缩放因子，最后对这两部分求和。计算完成后，根据结果对变形模型上点 V 的位置进行更新。

第三步：重复进行第二步，直至变形模型上所有点的位置都完成更新。

第四步：将列表中的信息更新为此次变化后（当前状态下）控制网格上各个面的信息，以便下次进入更新部分时计算与控制网格上各个面对应的缩放因子。



4.3 其他外包框变形坐标

1. 调和坐标

除了均值坐标和格林坐标外，还可以定义其他类似的外包框变形坐标。这些坐标由于所

计算的公式不同，都有自己的特点。例如，调和坐标和非负均值坐标。

调和坐标 (Harmonic Coordinate) 是另一种和均值坐标、格林坐标类似的基于外包框的坐标。调和坐标是一种由拉普拉斯 (Laplace) 方程推导出的坐标。由于拉普拉斯方程的解是调和函数，因此这种坐标称为调和坐标 (Harmonic Coordinate)。基于该坐标的变形称为调和变形 (Harmonic Deformation)。调和坐标相较于均值坐标有所改善的地方是在控制网格内部，保证任意点都是非负的，在变形过程中减少了操作者预期外的扭曲，同时它对控制网格内部点的影响随着距离而平稳衰减。此外，调和坐标的构建可以在任意维度中进行，并且在必要时可以额外的扩充内部的点、边和面，提供更细致的控制。调和坐标只定义在控制网格内部，调和坐标并不存在闭式表达式 (Closed - Form Expression)，也就是调和坐标的计算不能通过公式得出，只能通过数值计算得出。所以，调和坐标对于内存的需求也远大于均值坐标，这一点在三维情况下表现更为明显。

2. 非负均值坐标

前一节讲述的均值坐标保持了几个很有用的性质，如线性、精度、内插值及平滑度。然而，以均值坐标作为表面变形方法，主要的缺陷在于，对于非凸的区域，并不一定能保证得到的结果是非负值。这将会导致在控制网格非凸的情况下的变形产生不合理的扭曲。非负均值坐标 (PMVC, Positive Mean Value Coordinate) 不同于均值坐标的是计算结果对于任意外包框都是无条件非负的。与均值坐标相比，非负均值坐标的优点在于：在某些情况下，当控制网格上的两个分支相距较短的距离时，非负的均值坐标可以表现出更好的变形效果，减少模型在变形过程中表面上发生的扭曲。另一方面，除了模型不平滑或是包围网格过于粗糙的情况，非负均值坐标的效果大体上与调和坐标接近，并且根据模型和控制外包框的复杂程度，相比调和坐标可以节约数倍到数十倍的时间。



5.1 邻接矩阵

一个离散的三维模型是光滑曲面的近似。对于光滑曲面的特点和性质，数学家已经做了大量的研究，得出很多重要的结论。这些结论都是用数学公式来表示。一般来说，光滑曲面上的数学公式表示形式都是微积分的形式。对于光滑的三维曲面来说，表面上的每一个点周围都有无数个相邻的点。三维模型是离散的曲面，每个顶点只和有限数量的点相邻，同时还有边和面的元素。对于离散的三维模型，为了表示点、边、面之间的关系，可以用矩阵形式来表示。三维模型上的各种处理操作，如光滑、细分、变形、简化、展平等都是对于离散的曲面的操作。三维模型处理的算法就是实现这些操作的算法。这些离散三维模型处理的算法研究不是从头开始、凭空而来，都是借鉴于光滑曲面上数学家已经研究过的性质和结论。都要符合离散三维模型在逼近于光滑曲面的时候要满足光滑曲面上相应的数学公式。但是光滑曲面上的数学公式不能直接应用于离散的曲面，因为光滑曲面上的公式只能在光滑的情况下成立。因此对于离散的三维模型进行操作和处理，不但要借鉴、参考、依赖、受限制于光滑曲面的理论，而且还要把这些理论和公式进行离散化。把它们化作离散的形式，也就是矩阵的形式来计算。因此各种各样的三维模型处理操作最终都可以构建为矩阵和向量的运算。矩阵和线性系统是各种三维模型处理算法的核心。

邻接矩阵指的是表示三维模型上顶点、边和面之间互相关联的矩阵。也就是如果一个三维模型有 V 个顶点、 E 个边、 F 个面。那么顶点和面的邻接矩阵就是 $V \times F$ 个元素。如果两个元素相邻，那么相应的元素就是 1，不相邻相应的元素就是 0。这个邻接矩阵大部分元素都是零，因此需要用系数矩阵的方式来存储。

例如，图 5-1 表示了一个三维模型局部片段的顶点和顶点的邻接矩阵。

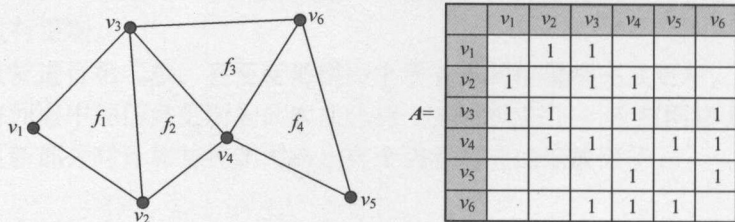


图 5-1 图形和矩阵

邻接矩阵根据所要表示的相邻元素类别的不同，可以分为顶点和顶点邻接、顶点和边邻接、顶点和面邻接等。

1. 顶点和顶点邻接矩阵

这个矩阵表示的是顶点和顶点的邻接关系，也就是矩阵的每一行表示一个顶点，每一列也表示一个顶点。一个三维模型如果有 V 个顶点，那么这个矩阵的大小是 $V \times V$ 。如果矩阵的顶点 i 和顶点 j 相邻，那么这个矩阵相应的第 i 行第 j 列的元素值为 1，其他的元素为零。这个矩阵里的元素可以表示为：

$$W_{ij} = \begin{cases} 1 & \text{if } (i, j) \in E \\ 0 & \text{其他} \end{cases}$$

输入一个三维模型，构建顶点和顶点邻接矩阵的代码如下：

```
public SparseMatrix BuildAdjacentMatrixVV( TriMesh mesh)
{
    SparseMatrix m = new SparseMatrix( mesh. Vertices. Count,
                                         mesh. Vertices. Count);
    foreach ( var center in mesh. Vertices)
    {
        foreach ( var round in center. Vertices)
        {
            m. AddElementIfNotExist( center. Index,
                                     round. Index, 1. 0);
        }
    }
    m. SortElement();
    return m;
}
```

2. 度数矩阵 (Degree Matrix)

这个矩阵也表示的是顶点和顶点邻接，但是和上述矩阵不同的地方在于如果两个顶点相连，那么相对应的元素的值不是 1，而是此顶点的度数，也就是此顶点所有相邻顶点的数量。

$$D_{ij} = \begin{cases} d_i = |N(i)| & i = j \\ 0 & \text{其他} \end{cases}$$

输入一个三维模型，构建度数的代码如下：

```
public SparseMatrix BuildMatrixDegree( TriMesh mesh)
{
    SparseMatrix m = new SparseMatrix( mesh. Vertices. Count,
                                         mesh. Vertices. Count);
    foreach ( TriMesh. Vertex vertex in mesh. Vertices)
    {
        m. AddElementIfNotExist( vertex. Index,
                                 vertex. Index, vertex. VertexCount);
    }
}
```

```

    }
    m.SortElement();
    return m;
}

```

3. 面和顶点的邻接矩阵

这个矩阵表示的是面和顶点相邻的矩阵，如果一个三维模型的三角形面数量为 F 、顶点数量为 V ，那么这个矩阵的大小为 $F \times V$ 。如果第 i 个面和第 j 个顶点相邻，那么这个矩阵第 i 行第 j 列的元素值为 1，其他的元素值为 0。

输入一个三维模型，构建面和顶点邻接矩阵的代码如下：

```

public SparseMatrix BuildAdjacentMatrixFV(TriMesh mesh)
{
    SparseMatrix m = new SparseMatrix(mesh.Faces.Count,
                                       mesh.Vertices.Count);
    foreach (var face in mesh.Faces)
    {
        foreach (var v in face.Vertices)
        {
            m.AddElementIfNotExist(face.Index, v.Index, 1.0);
        }
    }
    m.SortElement();
    return m;
}

```

4. 面和面的邻接矩阵

这个矩阵表示的是面和面相邻的矩阵，如果一个三维模型有 F 个三角形面，那么这个矩阵的大小为 $F \times F$ 。如果第 i 个面和第 j 个面相邻，那么这个矩阵第 i 行第 j 列的元素为 1，其他元素为 0。

输入一个三维模型，构建面和面的邻接矩阵的代码如下：

```

public SparseMatrix BuildAdjacentMatrixFF(TriMesh mesh)
{
    SparseMatrix m = new SparseMatrix(mesh.Faces.Count,
                                       mesh.Faces.Count);

    foreach (var center in mesh.Faces)
    {
        foreach (var round in center.Faces)
        {
            m.AddElementIfNotExist(center.Index, round.Index, 1.0);
        }
    }
}

```

```

    }
}
m.SortElement();
return m;
}

```

5. 顶点和边的邻接矩阵

这个矩阵表示顶点和边的相邻关系。如果一个三维模型有 V 个顶点、 E 个边，那么这个矩阵的大小为 $V \times E$ 。如果第 i 个顶点和第 j 个边相邻，那么这个矩阵第 i 行第 j 列的元素为 1，其他元素为 0。

输入一个三维模型，构建顶点和边的邻接矩阵的代码如下：

```

public SparseMatrix BuildAdjacentMatrixVE(TriMesh mesh)
{
    SparseMatrix m = new SparseMatrix( mesh.Vertices.Count,
                                         mesh.Edges.Count );

    foreach ( var v in mesh.Vertices )
    {
        foreach ( var edge in v.Edges )
        {
            m.AddElementIfNotExist( v.Index, edge.Index, 1.0 );
        }
    }
    m.SortElement();
    return m;
}

```

6. 顶点和面的邻接矩阵

这个矩阵表示三维模型顶点和面的相邻关系。如果一个三维模型有 V 个顶点、 F 个面，那么这个矩阵的大小为 $V \times F$ 。如果第 i 个顶点和第 j 个面相邻，那么这个矩阵第 i 行第 j 列的元素为 1，其他元素为 0。

输入一个三维模型，构建顶点和面的邻接矩阵的代码如下：

```

public SparseMatrix BuildAdjacentMatrixVF(TriMesh mesh)
{
    SparseMatrix m = new SparseMatrix( mesh.Vertices.Count,
                                         mesh.Faces.Count );

    foreach ( var v in mesh.Vertices )
    {
        foreach ( var face in v.Faces )
        {
            m.AddElementIfNotExist( v.Index, face.Index, 1.0 );
        }
    }
}

```



```

    }
}

m.SortElement();
return m;
}

```

7. 边和边的邻接矩阵

这个矩阵表示边和边之间的相邻关系。如果一个三维模型有 E 条边，那么这个矩阵的大小为 $E \times E$ 。如果第 i 条边和第 j 条边相邻，那么这个矩阵第 i 行第 j 列的元素为 1，其他元素为 0。

输入一个三维模型，构建边和边的邻接矩阵的代码如下：

```

public SparseMatrix BuildAdjacentMatrixEE(TriMesh mesh)
{
    SparseMatrix m = new SparseMatrix(mesh.Edges.Count,
                                       mesh.Edges.Count);
    foreach (var center in mesh.Edges)
    {
        foreach (var v in new[] { center.Vertex0, center.Vertex1 })
        {
            foreach (var round in v.Vertices)
            {
                m.AddElementIfNotExist(center.Index, round.Index, 1.0);
            }
        }
    }
    m.SortElement();
    return m;
}

```

8. 边和顶点的邻接矩阵

这个矩阵表示边和顶点之间的相邻关系。如果一个三维模型有 V 个顶点、 E 条边，那么这个矩阵的大小为 $V \times E$ 。如果第 i 条边和第 j 个顶点相邻，那么这个矩阵第 i 行第 j 列的元素为 1，其他元素为 0。

输入一个三维模型，构建边和顶点的邻接矩阵的代码如下：

```

public SparseMatrix BuildAdjacentMatrixEV(TriMesh mesh)
{
    SparseMatrix m = new SparseMatrix(mesh.Edges.Count,
                                       mesh.Vertices.Count);
    foreach (var edge in mesh.Edges)
    {
        m.AddElementIfNotExist(edge.Index, edge.Vertex0.Index, 1.0);
    }
}

```

```

        m.AddElementIfNotExist(edge.Index, edge.Vertex1.Index, 1.0);
    }
    m.SortElement();
    return m;
}

```

9. 边和面的邻接矩阵

这个矩阵表示边和面之间的相邻关系。如果一个三维模型有 F 个面、 E 条边，那么这个矩阵的大小为 $E \times F$ 。如果第 i 条边和第 j 个面相邻，那么这个矩阵第 i 行第 j 列的元素为 1，其他元素为 0。

输入一个三维模型，构建边和面的邻接矩阵的代码如下：

```

public SparseMatrix BuildAdjacentMatrixEF(TriMesh mesh)
{
    SparseMatrix m = new SparseMatrix(mesh.Edges.Count,
                                       mesh.Faces.Count);
    foreach (var edge in mesh.Edges)
    {
        m.AddElementIfNotExist(edge.Index, edge.Face0.Index, 1.0);
        m.AddElementIfNotExist(edge.Index, edge.Face1.Index, 1.0);
    }
    m.SortElement();
    return m;
}

```

10. 面和边的邻接矩阵

这个矩阵表示面和边之间的相邻关系。如果一个三维模型有 F 个面、 E 条边，那么这个矩阵的大小为 $F \times E$ 。如果第 i 个面和第 j 条边相邻，那么这个矩阵第 i 行第 j 列的元素为 1，其他元素为 0。

输入一个三维模型，构建面和边的邻接矩阵的代码如下：

```

public SparseMatrix BuildAdjacentMatrixFE(TriMesh mesh)
{
    SparseMatrix m = new SparseMatrix(mesh.Faces.Count,
                                       mesh.Edges.Count);
    foreach (var face in mesh.Faces)
    {
        foreach (var edge in face.Edges)
        {
            m.AddElementIfNotExist(face.Index, edge.Index, 1.0);
        }
    }
    m.SortElement();
}

```

```
return m;
```



5.2 组合拉普拉斯矩阵

5.2.1 拉普拉斯矩阵介绍

矩阵不仅仅可以表示离散的三维模型上点、边、面之间简单的邻接关系，还可以表示更复杂的含义。例如，可以表示一个数学操作符。在离散的三维模型上，三维模型上的数值都是离散的，可以表示为一个向量，如如果每个顶点上有一个数值，那么一个有 V 个顶点的三维模型，所有顶点上的数值就可以表示为一个有 V 个元素的向量。通过矩阵和向量的运算，就可以把这个向量变为另一个向量，从而矩阵也就可以表示为三维模型上的操作符。拉普拉斯操作符是三维模型处理的一个基本操作符，也是最常用的一个操作符。

拉普拉斯操作符（Laplace Operator）是数学里函数上的一个重要的操作符，是一个二阶微分算子，定义为函数的梯度的散度。公式如下：

$$\Delta = \text{div grad} = \nabla \cdot \nabla = \sum_i \frac{\partial^2}{\partial x_i^2}$$

也即是给定一个函数 f ，拉普拉斯算子把此函数映射为另一个函数：

$$\Delta f = \sum_{i=1}^n \frac{\partial^2 f}{\partial x_i^2}$$

拉普拉斯算子本身在光滑的曲面上有严格的数学定义，因此具有一些特定的性质。在离散的情况下，根据逼近光滑情况的不同，可以有多种拉普拉斯算子的定义方法。其中下面是几种常用的方法，根据应用的不同而采用不同的方法。在离散的情况下，拉普拉斯算子是一个矩阵 $L = (a_{ij})$ 。如果给三维模型上每个顶点定义一个数值，此数值的集合就是一个离散的函数。那么拉普拉斯算子就把这个数值集合映射为另一个数值集合。也就是每个顶点的数值经过拉普拉斯算子后变为另一个数值。如果三维模型有 V 个顶点，把三维模型每个顶点上的函数值表示为一个维数为 V 的向量 F ，那么拉普拉斯矩阵是一个 $V \times V$ 的矩阵 L ，对三维模型上的离散函数进行拉普拉斯操作，就是把拉普拉斯矩阵和向量相乘 $L \times F$ ，得到另一个维数为 V 的向量。拉普拉斯矩阵的非零元素和顶点之间的邻接矩阵类似，也就是如果两点之间有一条边相连，那么这两点之间的权重为非零。对于一个三维模型来说，这个矩阵是稀疏的，每行平均只有 6 个非零值。

拉普拉斯矩阵每一行的定义为：

$$(Lf)_i = b_i^{-1} \sum_{j \in N(i)} w_{ij} (f_i - f_j)$$

每条边上的权重是对称的：

(1) 每行的元素之和为零。如果函数用向量形式表示为 f ， f 的每个元素值都一样，那么拉普拉斯矩阵 L 和函数 f 相乘得到的向量值为零。因此拉普拉斯矩阵每行的元素之和为零才能使上述公式成立。

(2) 因为 $Lf=0$ ，那么拉普拉斯矩阵 L 具有一个值为常数的特征向量和相对应的值为 0

的特征值。

(3) 如果三维模型的组成部分为 c 个互不连接的模块, 那么相应的拉普拉斯矩阵有 c 个特征是为 0。

(4) 拉普拉斯矩阵可以表示为一个对角矩阵和一个对称矩阵的乘积:

$$L = B^{-1}S$$

其中, B^{-1} 是一个对角矩阵, 它的元素是 b_i^{-1} ; S 是一个对称矩阵, 对角线上的元素是 $S_{ii} = \sum_{j \in N(i)} w_{ij}$, 非对角线上的元素是 $-w_{ij}$ 。

(5) 虽然拉普拉斯矩阵 L 本身不是对称的, 但是 L 和对称矩阵 O 类似

$$O = B^{-1/2}SB^{-1/2}$$

因为

$$L = B^{-1}S = B^{-1/2}B^{-1/2}SB^{-1/2}B^{1/2} = B^{-1/2}OB^{1/2}$$

(6) L 和 O 有相同特征值, 假设 v 和 λ 是 O 的特征向量和特征值, 那么矩阵 L 的特征值为 λ , 特征向量为

$$u = B^{-1/2}v$$

(7) 对称矩阵 O 的特征向量是互相正交的, 但是拉普拉斯矩阵 L 的特征向量不是相互正交的。

(8) 虽然拉普拉斯矩阵的特征向量不是互相正交的, 但是在广义的含义下是正交的, 也就是

$$\langle u_i, u_j \rangle_B = u_i^T B u_j = v_i^T v_j = \delta_{ij}$$

(9) 如果权重 w_{ij} 是非负值, 那么拉普拉斯矩阵一定是半正定 (positive semi-definite) 的。

(10) 假如 b_i 的值都一样, 那么拉普拉斯矩阵是对称的。

拉普拉斯矩阵有很多种类, 根据是否考虑三维模型的拓扑信息和几何信息分为组合 (Combinatorial) 拉普拉斯矩阵和几何 (Geometric) 拉普拉斯矩阵两大类。如果只考虑三维模型的拓扑信息, 那么相应的拉普拉斯矩阵称为组合拉普拉斯矩阵。如果还考虑三维模型的几何信息, 那么相应的拉普拉斯矩阵称为几何拉普拉斯矩阵。在光滑的曲面上, 只有唯一的一个拉普拉斯操作符, 而在离散的三维模型上之所以有各种各样的拉普拉斯矩阵, 是由于拉普拉斯矩阵是光滑拉普拉斯操作符的离散化形式, 针对三维模型上某个特定的操作, 采用不同的离散形式, 能够得到更准确的计算结果。

5.2.2 拉普拉斯矩阵构建

组合拉普拉斯矩阵的形式为:

$$\begin{cases} a_{i,j} = w_{i,j} > 0 & \text{当}(i,j) \text{是边} \\ a_{i,j} = - \sum w_{i,j} \\ a_{i,j} = 0 & \text{其他} \end{cases}$$

其中, $w_{ij} = w_{ji}$, 根据权重的 w_{ij} 的不同, 有各种拉普拉斯矩阵。组合拉普拉斯矩阵的构建不考虑每个顶点的位置、三角形的面积和边长。只考虑顶点、边、面的拓扑邻接关系, 根据计算需要的不同, 有如下几种常用的构建方式。

1. 基于度数的组合拉普拉斯矩阵

这种拉普拉斯矩阵也称为图形拉普拉斯矩阵，矩阵相应的元素和顶点的度数相关。根据顶点邻接矩阵：

$$W_{ij} = \begin{cases} 1 & \text{if } (i,j) \in E \\ 0 & \text{其他} \end{cases}$$

和度数矩阵：

$$D_{ij} = \begin{cases} d_i = |N(i)| & \text{if } i=j \\ 0 & \text{其他} \end{cases}$$

可以得到图形拉普拉斯矩阵：

$$K = D - W$$

从三维模型得到图形拉普拉斯矩阵构建的代码如下：

```
public SparseMatrix BuildLaplaceGraph( TriMesh mesh)
{
    SparseMatrix vv = BuildAdjacentMatrixVV( mesh );
    SparseMatrix degree = BuildMatrixDegree( mesh );
    SparseMatrix graph = degree. Minus( vv );
    graph. SortElement( );
    return graph;
}
```

例如，图 5-2 左所示的简单网格的图形拉普拉斯矩阵如图 5-2 右所示。

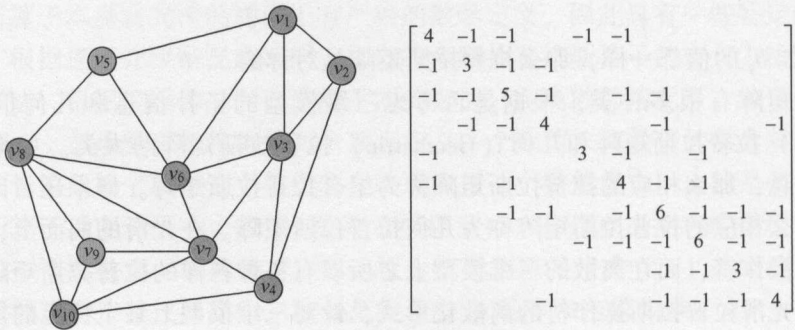


图 5-2 图形拉普拉斯矩阵

2. Tutte 拉普拉斯矩阵

Tutte 是另一种组合拉普拉斯矩阵，Tutte 拉普拉斯矩阵不是一个对称矩阵。这种拉普拉斯矩阵的定义为：

$$T = D^{-1}K$$

也就是矩阵的每个元素为：

$$T_{ij} = \begin{cases} 1 & \text{if } i=j \\ -1/d_i & \text{if } (i,j) \in E \\ 0 & \text{其他} \end{cases}$$

从三维模型得到 Tutte 拉普拉斯矩阵构建代码如下所示。

```
public SparseMatrix BuildLaplaceTutte( TriMesh mesh)
{
    SparseMatrix m = new SparseMatrix( mesh. Vertices. Count,
                                         mesh. Vertices. Count);
    foreach ( TriMesh. Vertex vertex in mesh. Vertices)
    {
        foreach ( TriMesh. Vertex neigh in vertex. Vertices)
        {
            double result = - 1d/vertex. VertexCount;
            m. AddElementIfNotExist( vertex. Index,
                                    neigh. Index,
                                    result);
        }
        m. AddValueTo( vertex. Index, vertex. Index, 1);
    }
    m. SortElement();
    return m;
}
```

3. 归一化图形拉普拉斯矩阵

还有一种拉普拉斯矩阵是归一化的拉普拉斯矩阵，这种矩阵对角线上的元素是 1，其他元素和两个相邻顶点的度数有关。这种矩阵的定义为：

$$Q = D^{-1/2} K D^{-1/2}$$

也就是矩阵的每个元素表示为：

$$Q_{ij} = \begin{cases} 1 & \text{if } i=j \\ -1/\sqrt{d_i d_j} & \text{if } (i,j) \in E \\ 0 & \text{其他} \end{cases}$$

这个拉普拉斯矩阵也不是一个标准意义上的拉普拉斯矩阵，不满足每行元素的和为零。从三维模型得到归一化的拉普拉斯矩阵构建代码如下：

```
public SparseMatrix BuildLaplaceGraphNomalized( TriMesh mesh)
{
    SparseMatrix m = new SparseMatrix( mesh. Vertices. Count,
                                         mesh. Vertices. Count);
    foreach ( TriMesh. Vertex vertex in mesh. Vertices)
    {
        foreach ( TriMesh. Vertex neigh in vertex. Vertices)
        {
            double result = - 1d/Math. Sqrt( vertex. VertexCount
                                             * neigh. VertexCount);
            m. AddValueTo( vertex. Index,
```



```
neigh. Index,
result);
}
m. AddValueTo(vertex. Index, vertex. Index, 1);
}
m. SortElement();
return m;
}
```

例如，对于三棱锥和立方体来说，这两个拉普拉斯矩阵的值如图 5-3 所示。

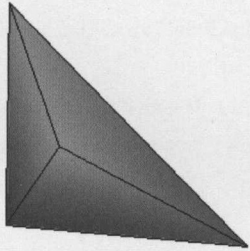
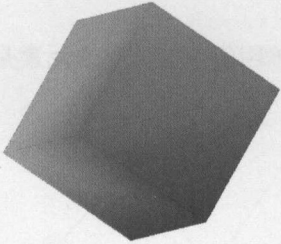
图形	图形拉普拉斯矩阵	Tutte拉普拉斯矩阵
	$\begin{bmatrix} 3 & -1 & -1 & -1 \\ -1 & 3 & -1 & -1 \\ -1 & -1 & 3 & -1 \\ -1 & -1 & -1 & 3 \end{bmatrix}$	$\begin{bmatrix} 1 & -0.33 & -0.33 & -0.33 \\ -0.33 & 1 & -0.33 & -0.33 \\ -0.33 & -0.33 & 1 & -0.33 \\ -0.33 & -0.33 & -0.33 & 1 \end{bmatrix}$
	$\begin{bmatrix} 5 & -1 & -1 & -1 & -1 & -1 & 0 & 0 \\ -1 & 4 & 0 & -1 & 0 & -1 & 0 & -1 \\ -1 & 0 & 4 & -1 & -1 & 0 & -1 & 0 \\ -1 & -1 & -1 & 5 & 0 & 0 & -1 & -1 \\ -1 & 0 & -1 & 0 & 5 & -1 & -1 & -1 \\ -1 & -1 & 0 & 0 & -1 & 4 & 0 & -1 \\ 0 & 0 & -1 & -1 & -1 & 0 & 4 & -1 \\ 0 & -1 & 0 & -1 & -1 & -1 & -1 & 5 \end{bmatrix}$	$\begin{bmatrix} 1 & -0.2 & -0.2 & -0.2 & -0.2 & -0.2 & 0 & 0 \\ -0.25 & 1 & 0 & -0.25 & 0 & -0.25 & 0 & -0.25 \\ -0.25 & 0 & 1 & -0.25 & -0.25 & 0 & -0.25 & 0 \\ -0.2 & -0.2 & -0.2 & 1 & 0 & 0 & -0.2 & -0.2 \\ -0.2 & 0 & -0.2 & 0 & 1 & -0.2 & -0.2 & -0.2 \\ -0.25 & -0.25 & 0 & 0 & -0.25 & 1 & 0 & -0.25 \\ 0 & 0 & -0.25 & -0.25 & -0.25 & 0 & 1 & -0.25 \\ 0 & -0.2 & 0 & -0.2 & -0.2 & -0.2 & -0.2 & 1 \end{bmatrix}$

图 5-3 两种矩阵示例



5.3 余切拉普拉斯矩阵

1. 计算公式

组合拉普拉斯矩阵只考虑三维模型顶点直接的链接关系，也就是拓扑关系，而不考虑每个顶点的位置，也就是不考虑几何信息。几何拉普拉斯矩阵不仅仅考虑三维模型的拓扑信息，还需要考虑三维模型的几何信息。最后得到的拉普拉斯矩阵也称为余切拉普拉斯矩阵。如果两个三维模型的拓扑一样，而顶点位置不一样，也就是外观不一样，那么这两个三维模型的组合图形拉普拉斯矩阵是一样的。在这种情况下，采用组合拉普拉斯矩阵就可能造成在某些需要三维模型处理操作上的失败。如果把每个顶点的位置考虑进去，可以得到余切拉普莱斯矩阵，这个矩阵也称为 Laplace - Beltrami 矩阵，这个矩阵是三维几何模型处理中最重要的一个矩阵，可以通过有限元或者外积分的方式推导出来。

余切拉普拉斯矩阵的权重是：

$$w_{ij} = \begin{cases} \frac{1}{2} (\cot \alpha_{ij} + \cot \alpha_{ji}) & \text{内部边缘} \\ \frac{1}{2} \cot \alpha_{ij} & \text{边界边缘} \end{cases}$$

其中, α_{ij}, α_{ji} 是两个和边 (x_i, x_j) 相对的角度, 如图 5-4 所示。

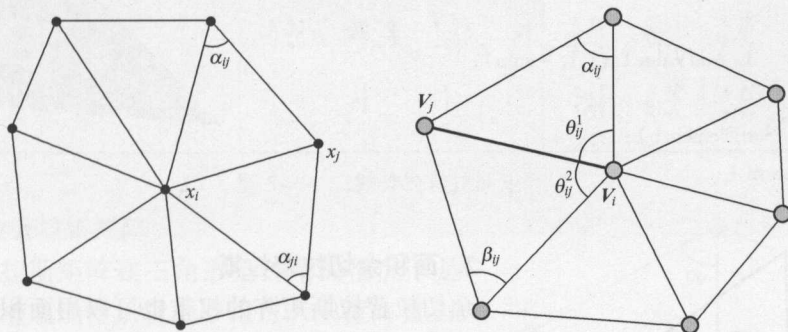


图 5-4 余切计算图示

在上述公式中, 每个矩阵元素的值需要计算两个角度的余切。在代码中, 不直接计算每个矩阵的元素, 而是首先构造一个 $V \times V$ 的矩阵, 遍历每个三角形, 计算三角形角度的余切, 把此余切值和到相应的矩阵元素中的值相加。遍历完后, 每个矩阵的元素正好就是两个对应角度余切的值之和。

余切拉普拉斯矩阵构建代码如下:

```
public SparseMatrix BuildLaplaceCot(TriMesh mesh)
{
    int n = mesh.Vertices.Count;
    SparseMatrix L = new SparseMatrix(n, n);
    for (int i = 0; i < mesh.Faces.Count; i++)
    {
        int c1 = mesh.Faces[i].GetVertex(0).Index;
        int c2 = mesh.Faces[i].GetVertex(1).Index;
        int c3 = mesh.Faces[i].GetVertex(2).Index;
        Vector3D v1 = mesh.Faces[i].GetVertex(0).Traits.Position;
        Vector3D v2 = mesh.Faces[i].GetVertex(1).Traits.Position;
        Vector3D v3 = mesh.Faces[i].GetVertex(2).Traits.Position;
        double cot1 = (v2 - v1).Dot(v3 - v1) / ((v2 - v1).Cross(v3 - v1).Length());
        double cot2 = (v3 - v2).Dot(v1 - v2) / ((v3 - v2).Cross(v1 - v2).Length());
        double cot3 = (v1 - v3).Dot(v2 - v3) / ((v1 - v3).Cross(v2 - v3).Length());
        L.AddValueTo(c1, c2, -cot3/2); L.AddValueTo(c2, c1, -cot3/2);
        L.AddValueTo(c2, c3, -cot1/2); L.AddValueTo(c3, c2, -cot1/2);
        L.AddValueTo(c3, c1, -cot2/2); L.AddValueTo(c1, c3, -cot2/2);
    }
    for (int i = 0; i < n; i++)
```

```

    {
        double sum = 0;
        foreach (SparseMatrix.Element e in L.Rows[i])
        {
            sum += e.value;
        }
        L.AddValueTo(i, i, -sum);
    }
    L.SortElement();
    return L;
}
    
```

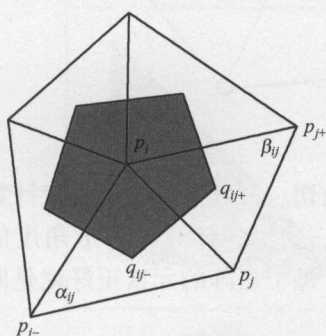


图 5-5 Voronoi 面积图示

2. 面积余切拉普拉斯

余切拉普拉斯矩阵的权重也可以用面积进行规则化，这样的拉普拉斯矩阵就不是对称的矩阵。计算公式如下所示：

$$\frac{1}{|\Omega_i|} \sum_{j \in N(i)} \frac{1}{2} (\cot \alpha_{ij} + \cot \beta_{ji}) (f_i - f_j)$$

其中的面积是 Voronoi 面积，如图 5-5 所示。

面积余切拉普拉斯矩阵的构建代码如下：

```

public SparseMatrix BuildLaplaceCotArea(TriMesh mesh)
{
    double[] areas = TriMeshUtil.ComputeAreaMixed(mesh);
    SparseMatrix cot = BuildLaplaceCot(mesh);
    int n = mesh.Vertices.Count;
    for (int i = 0; i < n; i++)
    {
        foreach (SparseMatrix.Element e in cot.Rows[i])
        {
            e.value = e.value * (1/areas[i]);
        }
    }
    return cot;
}
    
```

三棱锥拉普拉斯矩阵如图 5-6 所示。

在各种拉普拉斯矩阵中，最重要的是余切拉普拉斯矩阵，各种三维模型处理算法，如变形、分段、光滑等都用到余切拉普拉斯矩阵。在有些算法和模型处理应用的时候，余切拉普拉斯矩阵需要乘以 -1。

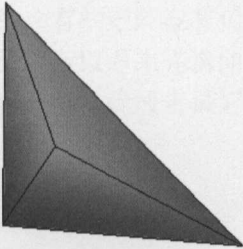
图形	余切拉普拉斯矩阵				面积余切拉普拉斯矩阵			
	1.57	-0.28	-0.28	-1	2.92	-0.53	-0.53	-1.85
	-0.28	1.57	-0.28	-1	-0.53	2.92	-0.53	-1.85
	-0.28	-0.28	1.57	-1	-0.53	-0.53	2.92	-1.85
	-1	-1	-1	3	-2.66	-2.66	-2.66	8

图 5-6 三棱锥拉普拉斯矩阵

3. 中值拉普拉斯矩阵

余切拉普拉斯矩阵在三角形是钝角的情况下是负值，这样会造成线性系统的不稳定。中值（Mean Value）拉普拉斯矩阵采用如下的权重，从而矩阵所有元素的值都是正值。

$$w_{ij} = \frac{\tan(\theta_{ij}^1/2) + \tan(\theta_{ij}^2/2)}{\|v_i - v_j\|}$$

公式的角度如图 5-7 所示。

中值拉普拉斯矩阵的构建代码如下：

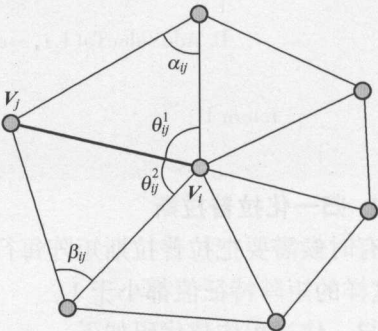


图 5-7 公式的角度

```
public SparseMatrix BuildMatrixMeanValue(TriMesh mesh)
{
    int n = mesh.Vertices.Count;
    SparseMatrix L = BuildAdjacentMatrixVV(mesh);
    SparseMatrix D = new SparseMatrix(n,n);
    foreach (TriMesh.HalfEdge halfedge in mesh.HalfEdges)
    {
        TriMesh.Vertex fromJ = halfedge.FromVertex;
        TriMesh.Vertex toI = halfedge.ToVertex;
        Vector3D jtoI = (toI.Traits.Position - fromJ.Traits.Position).Normalize();
        Vector3D alphaToI = (halfedge.Next.ToVertex.Traits.Position
            - toI.Traits.Position).Normalize();
        Vector3D betaToI = (halfedge.Opposite.Previous.FromVertex.Traits.Position
            - toI.Traits.Position).Normalize();
        double cosGamaIJ = jtoI.Dot(alphaToI)/(jtoI.Length()*alphaToI.Length());
        double cosThetaIJ = jtoI.Dot(betaToI)/(jtoI.Length()*betaToI.Length());
        double angelGama = Math.Acos(cosGamaIJ);
        double angelTheta = Math.Acos(cosThetaIJ);
        double wij = (Math.Tan(angelGama/2) + Math.Tan(angelTheta/2))
            /(toI.Traits.Position - fromJ.Traits.Position).Length();
```

```

        int indexI = toI.Index;
        int indexJ = fromJ.Index;
        D.AddValueTo(indexI, indexJ, wij);
    }
    for (int i = 0; i < n; i++)
    {
        double sum = 0;
        foreach (SparseMatrix.Element item in D.Rows[i])
        {
            sum += item.value;
        }
        D.AddValueTo(i, i, -sum);
    }
    return D;
}

```

4. 归一化拉普拉斯

有时候需要把拉普拉斯矩阵每行的元素值归一化，也就是对角线为 1，其他值之和等于 1。这样的矩阵特征值都小于 1。

归一化过程构建代码如下：

```

public SparseMatrix BuildMatrixCotNormalize(TriMesh mesh)
{
    int n = mesh.Vertices.Count;
    SparseMatrix L = BuildLaplaceMatrixCotBasic(mesh);
    for (int i = 0; i < n; i++)
    {
        double sum = 0;
        foreach (SparseMatrix.Element e in L.Rows[i])
        {
            sum += e.value;
        }
        foreach (SparseMatrix.Element e in L.Rows[i])
        {
            e.value = e.value/sum;
        }
        L.AddValueTo(i, i, -1);
    }
    L.SortElement();
    return L;
}

```

这些拉普拉斯矩阵在各种三维模型处理中都会用到,根据应用的不同,采用不同的拉普拉斯矩阵,需要根据特定的操作进行分析所需要的矩阵属性。在这些拉普拉斯矩阵中,有些矩阵是对称的,有些矩阵没有负值,有些矩阵对角线是1。因为三位模型是离散的模型,不是光滑的曲面,所以原来光滑表面上的拉普拉斯算子离散化后就有不同的方式,这些方式一般来说没有哪个是最优或者最好,只有根据具体的操作来决定采用哪种矩阵。



图 5-1-1 离散化后的三维模型

图 5-1-1 离散化后的三维模型

图 5-1-1 离散化后的三维模型

图 5-1-1 离散化后的三维模型

图 5-1-1 离散化后的三维模型

图 5-1-1 离散化后的三维模型

图 5-1-1 离散化后的三维模型

图 5-1-1 离散化后的三维模型

图 5-1-1 离散化后的三维模型

图 5-1-1 离散化后的三维模型

图 5-1-1 离散化后的三维模型

图 5-1-1 离散化后的三维模型

图 5-1-1 离散化后的三维模型

图 5-1-1 离散化后的三维模型

图 5-1-1 离散化后的三维模型

图 5-1-1 离散化后的三维模型

图 5-1-1 离散化后的三维模型

图 5-1-1 离散化后的三维模型

图 5-1-1 离散化后的三维模型

图 5-1-1 离散化后的三维模型

图 5-1-1 离散化后的三维模型

图 5-1-1 离散化后的三维模型

图 5-1-1 离散化后的三维模型

图 5-1-1 离散化后的三维模型

图 5-1-1 离散化后的三维模型

图 5-1-1 离散化后的三维模型

图 5-1-1 离散化后的三维模型

图 5-1-1 离散化后的三维模型

图 5-1-1 离散化后的三维模型

图 5-1-1 离散化后的三维模型

图 5-1-1 离散化后的三维模型

图 5-1-1 离散化后的三维模型

图 5-1-1 离散化后的三维模型

图 5-1-1 离散化后的三维模型

图 5-1-1 离散化后的三维模型

图 5-1-1 离散化后的三维模型

图 5-1-1 离散化后的三维模型

图 5-1-1 离散化后的三维模型

图 5-1-1 离散化后的三维模型

图 5-1-1 离散化后的三维模型

图 5-1-1 离散化后的三维模型

图 5-1-1 离散化后的三维模型

图 5-1-1 离散化后的三维模型

图 5-1-1 离散化后的三维模型

图 5-1-1 离散化后的三维模型

图 5-1-1 离散化后的三维模型

图 5-1-1 离散化后的三维模型

图 5-1-1 离散化后的三维模型

图 5-1-1 离散化后的三维模型

图 5-1-1 离散化后的三维模型

图 5-1-1 离散化后的三维模型

图 5-1-1 离散化后的三维模型

图 5-1-1 离散化后的三维模型

图 5-1-1 离散化后的三维模型

图 5-1-1 离散化后的三维模型

图 5-1-1 离散化后的三维模型

图 5-1-1 离散化后的三维模型

图 5-1-1 离散化后的三维模型

图 5-1-1 离散化后的三维模型

图 5-1-1 离散化后的三维模型

图 5-1-1 离散化后的三维模型

图 5-1-1 离散化后的三维模型

图 5-1-1 离散化后的三维模型

图 5-1-1 离散化后的三维模型

图 5-1-1 离散化后的三维模型

图 5-1-1 离散化后的三维模型

图 5-1-1 离散化后的三维模型

图 5-1-1 离散化后的三维模型

图 5-1-1 离散化后的三维模型

图 5-1-1 离散化后的三维模型



6.1 微分坐标

拉普拉斯变形算法的名称来源于拉普拉斯操作符矩阵。这个变形算法依赖拉普拉斯矩阵进行对模型的变形操作。通过一个三维模型可以构建一个拉普拉斯矩阵，通过拉普拉斯矩阵，可以得到每个顶点的局部信息。在保持局部信息不变的情况下，可以实现三维模型的变形。拉普拉斯变形算法的优点是能够保持三维模型的细节不发生改变，只改变三维模型宏观上的形状。这一点在变形上非常重要，如对于一个雕刻有花纹的瓶子，变形的时候想改变的是瓶子的长度，或者宽度，并不想改变瓶子上面的花纹。拉普拉斯变形算法在操作界面上也非常简单，不需要额外的工具，只需要对三维模型上选择的部分进行相应的移动，旋转就可以对整个三维模型进行变形。拉普拉斯变形算法构建为一个线性系统，因此求解过程非常快，可以实时交互地对三维模型进行变形。拉普拉斯变形算法已经是一个成熟的变形算法。

微分坐标 (differential coordinates) 是拉普拉斯变形算法的核心概念，是三维模型处理中一个重要的工具。和三维模型顶点的欧几里得坐标不一样，微分坐标是个局部坐标，包含了三维模型的局部信息，也就是包含了顶点和周围顶点之间的相对关系。微分坐标是三维模型的另一种表示方式。因为微分坐标定义在全局坐标系里面，因此微分坐标不是旋转不变的 (rotation - invariant)，也就是如果三维模型发生旋转，那么微分坐标也会发生相应的旋转。这一点是微分坐标一个非常重要的特征。微分坐标也称为拉普拉斯坐标。

1. 微分坐标的定义

指定 $G = (V, E)$ 表示一个三维模型，其中 V 是三维模型的所有顶点的集合， E 是三维模型的左右边的集合， V_i 表示一个顶点， V_j 表示此顶点的邻接顶点。那么微分坐标定义如下：

$$\delta_i = L(v_i) = v_i - \frac{1}{d_{i \in N(i)}} \sum v_j$$

微分坐标是定义在每个顶点上的，由这个顶点和这个顶点周围的邻居顶点的位置决定，如图 6-1 中所示。红色顶点表示一个顶点，其他顶点是此红色顶点的邻居顶点，那么黄色箭头就表示红色顶点的微分坐标。微分坐标是一个向量。

或者如图 6-2 和图 6-3 所示。

$$\delta_i = \frac{1}{d_{i \in N(i)}} \sum (v_i - v)$$

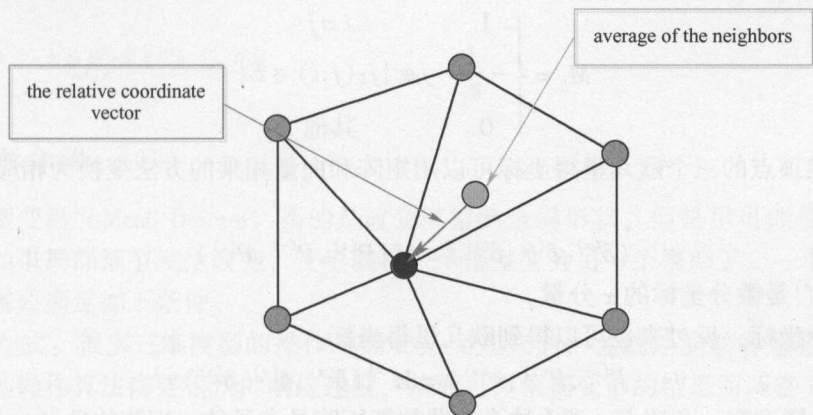


图 6-1 微分坐标图示 (1)

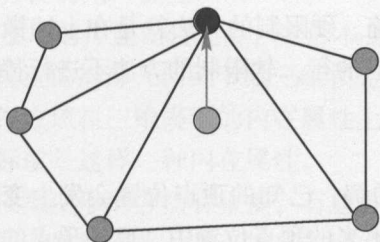


图 6-2 微分坐标图示 (2)

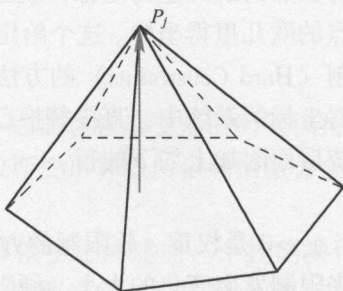


图 6-3 微分坐标图示 (3)

如图 6-4 所示, 离散三维模型上每个顶点的微分坐标是光滑曲面上点的微分坐标的近似, 在光滑曲面的情况下, 微分坐标的定义为:

$$\frac{1}{\text{len}(\gamma)} \int_{v \in \gamma} (v_i - v) ds$$

三维模型上定义的微分坐标就是这个顶点周围顶点位置的平均值。每个顶点的微分坐标是一个向量。这个向量的方向和顶点法向的方向一致, 这个向量的大小和顶点的平均曲率一致。

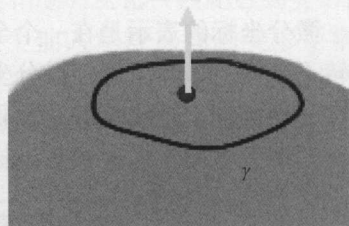


图 6-4 光滑曲面微分坐标

2. 坐标变换

三维模型在输入的时候每个顶点的坐标表示为欧几里得坐标, 具有 x 、 y 、 z 三个分量。每个顶点的 x 分量可以构成一个向量 \mathbf{X} , y 分量可以构成一个向量 \mathbf{Y} , z 分量可以构成一个向量 \mathbf{Z} 。如果一个三维模型有 V 个顶点, 那么这 3 个向量分别有 V 个元素。每个顶点的微分坐标也有 3 个分量, 也可以分别构成 3 个向量。从而整个三维模型的顶点欧几里得坐标可以用 \mathbf{X} 、 \mathbf{Y} 、 \mathbf{Z} 三个向量来表示。那么欧几里得坐标向量变换为微分几何坐标向量可以用一个大小为 $V \times V$ 的矩阵来表示。根据微分坐标的定义, 根据之前微分坐标的定义, 这个矩阵中的元素计算公式如下:

$$M_{ij} = \begin{cases} 1 & i=j \\ -\frac{1}{d_i} & j \in \{j:(j,i) \in E\} \\ 0 & \text{其他} \end{cases}$$

从而三维顶点的三个欧几里得坐标可以用矩阵和向量相乘的方法变换为相应的三个微分坐标：

$$(\delta^{(x)}, \delta^{(y)}, \delta^{(z)}) = M(P^{(x)}, P^{(y)}, P^{(z)})$$

其中， $\delta^{(x)}$ 是微分坐标的 x 分量。

根据微分坐标，反过来也可以得到欧几里得坐标：

$$(P^{(x)}, P^{(y)}, P^{(z)}) = M^{-1}(\delta^{(x)}, \delta^{(y)}, \delta^{(z)})$$

如果三维模型有 n 个顶点，那么这个拉普拉斯矩阵是奇异的，矩阵的秩（rank）是 $n - k$ 。其中， k 是三维模型的连通体数量。因此，如果已知微分坐标和拉普拉斯矩阵，需要得到三维模型所有顶点的欧几里得坐标，就必须给定大于 k 个顶点的欧几里得坐标，然后才能求解出其余顶点的欧几里得坐标。这个给定顶点的欧几里得坐标可以用软限制（Soft Constraints）或者硬限制（Hard Constraints）的方法加入上述系统。硬限制的方法就是在上述微分坐标求解欧几里得坐标的系统中，直接替换已知顶点相对应的行，软限制的方法不进行替换，而是在原来系统后面附加上如下的行：

$$w_i x_i = w_i c_i$$

其中， $w_i > 0$ 是权重。软限制的方法求解出来的解，已知的顶点位置会发生变化，通过这个权重来限制发生变化的大小。硬限制方法求解出来的顶点位置中，已知顶点的位置不会发生任何变化。

3. 旋转不变性

微分坐标的表示是在一个全局的坐标系里，是一个向量，具有方向，因此微分坐标不是旋转不变的。虽然可以把微分坐标表示在一个局部的坐标系，但是这样从欧几里得坐标转换为微分坐标就不是线性的了，而是非线性的，从而会给后续的计算带来麻烦。如图 6-5 所示，模型发生旋转后，微分坐标的方向也发生了相应的变化，但是大小保持不变。

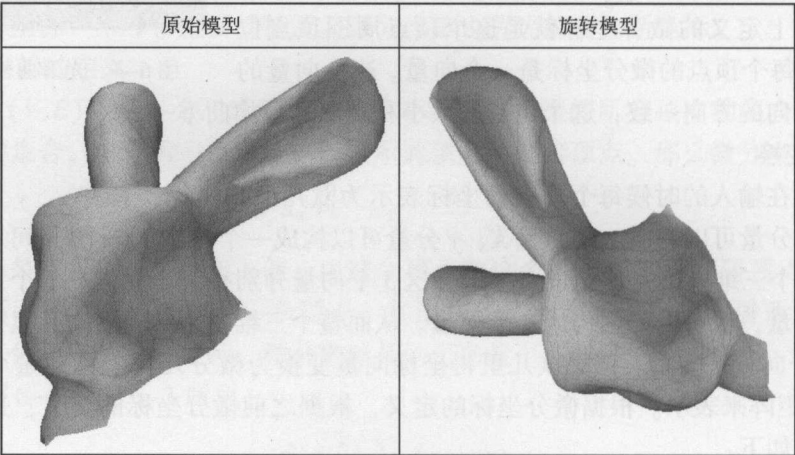


图 6-5 微分坐标旋转不变



6.2 变形算法基础

6.2.1 变形介绍

三维模型变形 (Mesh Deform) 指的是改变模型的全局形状, 但是尽可能保持局部细节不变。因为如果局部细节发生改变, 变形就把一个模型变为另一个模型了。一个好的三维模型变形算法需要满足如下条件。

(1) 交互式: 很多三维模型的操作不需要是交互式的, 也就是能够容忍较长的处理时间, 而变形的操作算法需要很快的响应速度, 从而可以根据变形的结果而调整变形。

(2) 变形的操作还应该能够保持模型的局部细节不发生变化, 而只改变模型的形状。也就是如果三维模型局部是光滑的, 或者具有某些特征, 那么这些光滑或者特征在变形后仍得到保持。

(3) 方便的用户交互。

(4) 变形的传递比较光滑, 变形本身也比较光滑。

三维模型的局部几何细节是三维模型的内在属性, 因此在模型变形的时候保持内在属性不变的操作应该在三维模型的内在属性上进行。拉普拉斯变形算法所依赖的微分坐标或者拉普拉斯坐标就是这样一种内在属性。

三维模型变形的有很多, 有些方法需要在三维模型之外建立一个控制操作的工具, 如外包框或者控制网格。基于拉普拉斯的变形方法, 只需要选定三维模型上的一些顶点作为固定不变的区域, 另一些顶点作为产生变形的区域, 这些区域称为变形手柄 (handle)。通过移动变形手柄, 其他没被选中的顶点通过拉普拉斯系统发生相应的变化, 因此达到变形的效果。例如, 在图 6-6 中, 仙人掌上有三块彩色的地方是这个仙人掌被选择的变形手柄, 图 6-6 右中马的身上四个脚和嘴巴部分是变形手柄。这些变形手柄在变形的时候, 可以固定任意一个或几个, 然后移动, 旋转其他的变形手柄。固定的和变化的手柄可以互相交换。

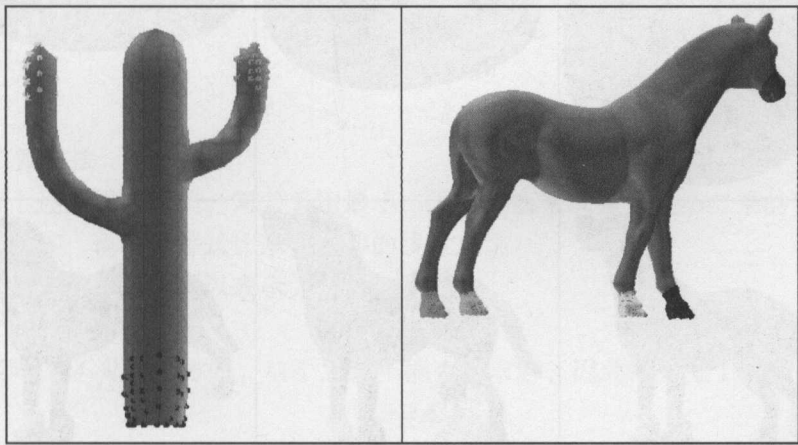


图 6-6 变形把柄

拉普拉斯算法可以作用在各种不同形状、不同拓扑的三维模型上, 并且都能够得到很好的变形效果, 对于用户的交互来说, 也能够很方便实现变形。在操作上, 最终的变形和用户

在操作时的预期变形效果基本一致。例如，如图 6-7 所示的几个三维模型的拉普拉斯变形效果。

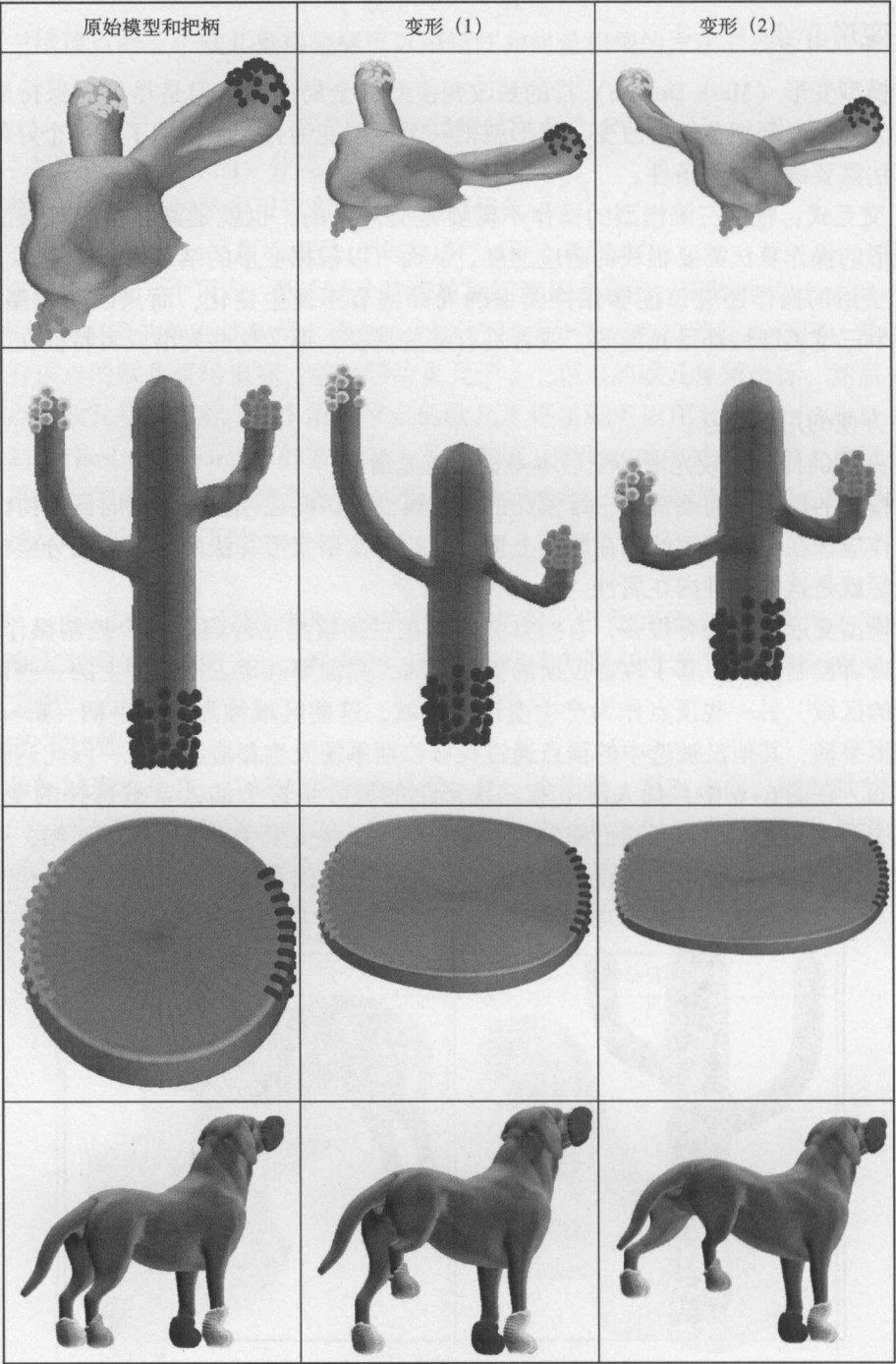


图 6-7 各种三维模型拉普拉斯变形示例

6.2.2 数学模型构建

1. 数学原理

拉普拉斯变形的核心理念是把顶点的坐标从欧几里得空间转变到微分坐标空间。因为微分坐标是个局部的坐标,从而保持微分坐标不变,就保持了三维模型的局部细节不变。进而在满足用户交互改变的三维模型一部分顶点的位置条件下,再把三维模型的微分坐标恢复为欧几里得坐标,从而完成整个变形过程。

第一步:计算每个顶点的微分坐标:

$$\Delta = L(V)$$

其中, V 是向量,分别对应 x 、 y 、 z 三个坐标轴的三个向量,表示三维模型顶点没有变形时的坐标; L 表示拉普拉斯操作符,是一个矩阵; Δ 表示得到的微分坐标,或者拉普拉斯坐标,可以是三个向量,分别对应三个坐标轴。

$$V = [v_1^T, v_2^T, \dots, v_n^T]^T$$

$$v_i = [v_{ix}, v_{iy}, v_{iz}]^T \in R^3$$

第二步:移动变形手柄,得到变形手柄上顶点新的欧几里得坐标:

$$v'_i = u_i, i \in C$$

其中, u_i 表示手柄顶点新位置; v'_i 表示第 i 个顶点的新的位置。

第三步:根据微分坐标和手柄上顶点的新位置,用最小二乘法计算其余顶点的位置:

$$V = \operatorname{argmin} \left(\|L(V') - \Delta\|^2 + \sum_{i \in C} \|v'_i - u_i\|^2 \right)$$

其中, V' 表示所有顶点的新位置向量。

第四步:矩阵形式为

$$AV' = b$$

其中,

$$A = \begin{pmatrix} L \\ F \end{pmatrix}, F_{ij} = \begin{cases} 1 & j = s_i \in C \\ 0 & \text{其他} \end{cases}$$

$$b_k = \begin{cases} 0 & k \leq n \\ x_{s_{k-n}} & n < k \leq n+m \end{cases}$$

2. 最小二乘法

为了求解上述的线性系统,需要采用最小二乘的方法来求解。最小二乘的方法是求解矩阵的行数比列数多,也就是等式比未知数多的线性系统。

第一步:上述优化问题可以表示为

$$\min \|Ax - b\|$$

其中,矩阵 A 不是一个方阵,这个系统不能直接求解,因此需要把上述系统变化为

$$A^T Ax = A^T b$$

最终的未知数的值为

$$x = (A^T A)^{-1} A^T b$$

求解矩阵的逆矩阵非常困难,因此用求解线性方程的方法求解。这个系统是个稀疏线性系统,因此可以很快求解。

第二步： $A^T A$ 是正定的，可以分解为

$$A^T A = R^T R$$

第三步：方程组变为

$$R^T R x = A^T b$$

其中， R 是上三角阵。

$A^T A$ 的分解是一个耗时的操作，但是这一步只需要进行一次。

第四步：每次手柄发生变化后，上述系统中只有 b 发生变化，而矩阵 A 不发生变化，因此不需要重新分解矩阵。

第五步：可以直接用之前分解后的上三角阵，然后采用后代入法（back - substitution）求得新的未知数。

第六步：用一个中间变量求解最终 x 。

$$R^T \tilde{x} = A^T b$$

$$R x = \tilde{x}$$

6.2.3 拉普拉斯变形算法代码

和上节的数学构建相对应的算法代码也可以分为构建拉普拉斯矩阵，增加限制条件，构建线性系统，用最小二乘法求解此线性系统，更新三维模型的顶点坐标等几个模块函数。虽然三维顶点的 x 、 y 、 z 三个坐标分量可以合并起来求解，但是这样，给定一个具有 V 个顶点的三维模型，线性系统是 $3V \times 3V$ ，非常大，因此在实现的时候，可以分开求解三个 $V \times V$ 的较小的线性系统，从而加快算法的执行。

第一步：构建拉普拉斯矩阵。

```
public virtual SparseMatrix BuildMatrixA()
{
    SparseMatrix A = LaplaceManager.Instance.BuildLaplaceTutte(mesh);
    return A;
}
```

第二步：给上述矩阵加入限制条件。

```
public virtual void BuildConstraints(SparseMatrix A)
{
    if (A == null)
        throw new Exception("A matrix is null");

    if (handlers != null)
    {
        for (int i = 0; i < handlers.Count; i++)
        {
            A.AddRow();
            A.AddElement(A.RowSize - 1, handlers[i],
                ConfigLaplace.Instance.HandleConstraintScale);
        }
    }
}
```

```

    }
}
else
{
    for (int i=0;i < mesh.Vertices.Count;i++)
    {
        if (mesh.Vertices[i].Traits.SelectedFlag != 0)
        {
            A.AddRow();
            A.AddElement(A.RowSize-1,i,
                ConfigLaplace.Instance.HandleConstraintScale);
        }
    }
}
A.SortElement();
}

```

第三步：把第二步得到的长方阵乘以转秩得到一个方阵。

```
SparseMatrix ATA = LaplaceManager.Instance.BuildMatrixATA(ref MatrixA,mesh);
```

第四步：分解这个方阵。

```
LinearSystem.Instance.Factorize(ref ATA);
```

第五步：计算拉普拉斯坐标。

```

public virtual double[][] BuildLaplacian()
{
    SparseMatrix L = LaplaceManager.Instance.BuildLaplaceTutte(mesh);
    double[][] lap = LaplaceManager.Instance.ComputeLaplacianBasic(L,mesh);
    return lap;
}

```

第六步：构建线性系统的上半部分。

```

public virtual double[][] BuildBUpPart()
{
    double[][] lap = BuildLaplacian();
    double[][] b = new double[3][];
    int n = MatrixA.RowSize;
    for (int i=0;i < 3;i++)
    {
        b[i] = new double[n];

        for (int j=0;j < mesh.Vertices.Count;j++)

```

```

        {
            b[i][j] = lap[i][j];
        }
    }

    lap = null;
    System.GC.Collect();
    return b;
}

```

第七步：为第六步构建的向量加入手柄位置的限制条件。

```

public virtual void UpdateHandle()
{
    int k = mesh.Vertices.Count;
    if (handlers != null)
    {
        for (int i = 0; i < handlers.Count; i++)
        {
            b[0][k] = mesh.Vertices[handlers[i]].Traits.Position.x
                * ConfigLaplace.Instance.HandleConstraintScale;
            b[1][k] = mesh.Vertices[handlers[i]].Traits.Position.y
                * ConfigLaplace.Instance.HandleConstraintScale;
            b[2][k] = mesh.Vertices[handlers[i]].Traits.Position.z
                * ConfigLaplace.Instance.HandleConstraintScale;
            k++;
        }
    }
    else
    {
        for (int i = 0; i < mesh.Vertices.Count; i++)
        {
            if (mesh.Vertices[i].Traits.SelectedFlag != 0)
            {
                b[0][k] = mesh.Vertices[i].Traits.Position.x
                    * ConfigLaplace.Instance.HandleConstraintScale;
                b[1][k] = mesh.Vertices[i].Traits.Position.y
                    * ConfigLaplace.Instance.HandleConstraintScale;
                b[2][k] = mesh.Vertices[i].Traits.Position.z
                    * ConfigLaplace.Instance.HandleConstraintScale;
                k++;
            }
        }
    }
}

```


第八步：用 A 矩阵的转置 A^T 和 b 计算线性方程组的右边部分。

```
ATb = MatrixA. Transpose( ). Multiply( b[0] );
```

第九步：求解这个线性方程组，得到变形后新的位置。分为 x 、 y 、 z 三个线性方程组。

```
public void ComputeDeform()
{
    ATb = MatrixA. Transpose( ). Multiply( b[0] );
    unknown[0] = LinearSystem. Instance. SolveByPreFactorize( ref ATb );
    ATb = MatrixA. Transpose( ). Multiply( b[1] );
    unknown[1] = LinearSystem. Instance. SolveByPreFactorize( ref ATb );
    ATb = MatrixA. Transpose( ). Multiply( b[2] );
    unknown[2] = LinearSystem. Instance. SolveByPreFactorize( ref ATb );
}
```

第十步：更新三维模型的顶点位置。

```
public void UpdateMesh()
{
    for ( int i=0; i < mesh. Vertices. Count; i++ )
    {
        mesh. Vertices[i]. Traits. Position. x = unknown[0][i];
        mesh. Vertices[i]. Traits. Position. y = unknown[1][i];
        mesh. Vertices[i]. Traits. Position. z = unknown[2][i];
    }
}
```



6.3 拉普拉斯变形迭代算法

6.3.1 迭代法介绍

基于微分坐标的拉普拉斯变形能够保持局部特征和增强对变形的控制。理想的情况下，微分坐标应该是旋转不变的，这样的话，可以确保变形时细节不会失真。但是因为微分坐标不是旋转不变的，因此会产生细节上的扭曲。在变形的过程中，手柄可以进行平移或者旋转，这两种情况都会造成其他顶点微分坐标的改变。因为即使是平移，在手柄位置变化的情况下，虽然手柄本身没有发生旋转，但是手柄部分和三维模型其余部分的相对位置发生了变化，因此也会产生旋转。

二维情况如图6-8所示，蓝色曲线是原始曲线，把曲线中间的一个顶点平移到上方，

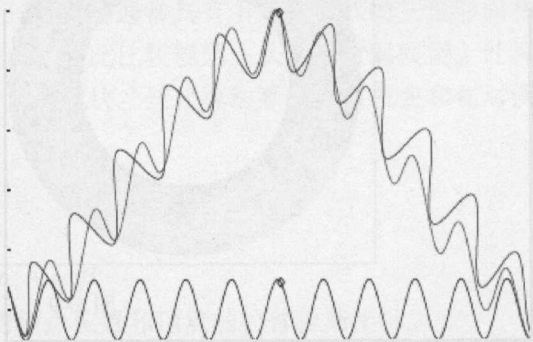


图6-8 一维曲线变形

那么绿色曲线是不考虑微分坐标旋转的结果，而红色曲线是考虑微分坐标旋转的结果，从中可以看出，红色曲线的效果更好。

由于微分坐标不是旋转不变的，因此在上述系统中，需要把微分坐标进行相应的旋转，如下列公式所示：

$$V = \arg \min \left(\sum_{i=1}^n \|L(v'_i) - T_i(\delta_i)\|^2 + \sum_{j \in C} \|v'_j - u_j\|^2 \right)$$

其中， T_i 表示微分坐标的旋转矩阵。把三维模型顶点的全局欧几里得坐标变换为拉普拉斯坐标，然后根据拉普拉斯坐标和若干移动后的顶点的欧几里得坐标重建三维模型是拉普拉斯变形方法的核心思想。但是这个变形方法存在一个难点，就是拉普拉斯坐标不是旋转不变的，也就是需要把原来的拉普拉斯坐标进行相应的旋转。这是一个鸡和蛋的问题，一方面，重建三维模型的顶点位置需要知道准确的拉普拉斯坐标旋转后的值，另一方面，拉普拉斯坐标的值和重建后的三维模型的顶点位置有关。解决这个问题的方法有很多，其中之一是插值法，也就是把手柄上顶点的拉普拉斯坐标旋转的角度进行插值到其他待定顶点的拉普拉斯坐标。但是插值法的缺点是对于变化比较大的变形和手柄位移产生的旋转情况效果不好。另一种解决拉普拉斯坐标更新的方法是迭代法，这种方法的思路是多次迭代求解，而不是依次解决问题。每次更新原来的坐标，经过多次迭代后达到最优值。

拉普拉斯坐标包含局部几何信息和局部参数化信息，是一个向量，不仅有大小还有方向。这个向量除了法向的分量外，还有一个切线方向的分量。迭代法的核心思想是保持拉普拉斯微分坐标的大小不变，但是改变微分坐标的方向。保持微分坐标的大小可以保持原有模型的局部形状，而每次改变拉普拉斯坐标向量的方向为顶点的当前法向方向，这样就可以移除切线方向的分量，从而减少变形产生的拉伸和扭曲。

迭代算法的拉普拉斯变形算法的效果相比上一节讲述的变形算法效果上要好，但是由于进行了多次迭代，因此速度要慢。在各种拓扑、形状的三维模型上进行拉普拉斯迭代算法变形的效果如图 6-9 所示。从图中可以看出，对于把柄平移所引起的旋转操作，可以平滑地过度到三维模型的其余部分。

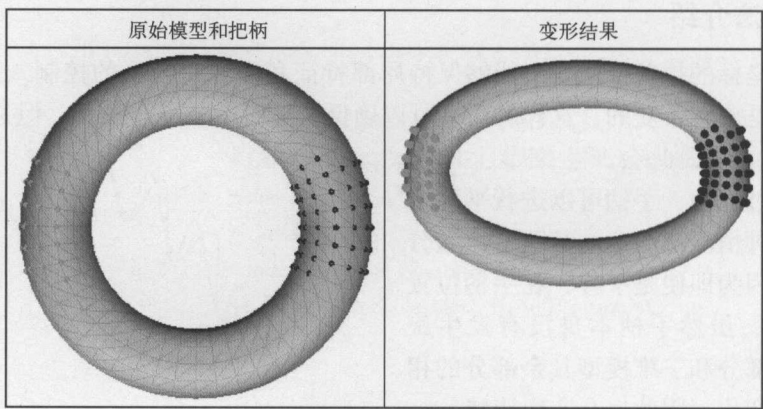


图 6-9 三维模型变形

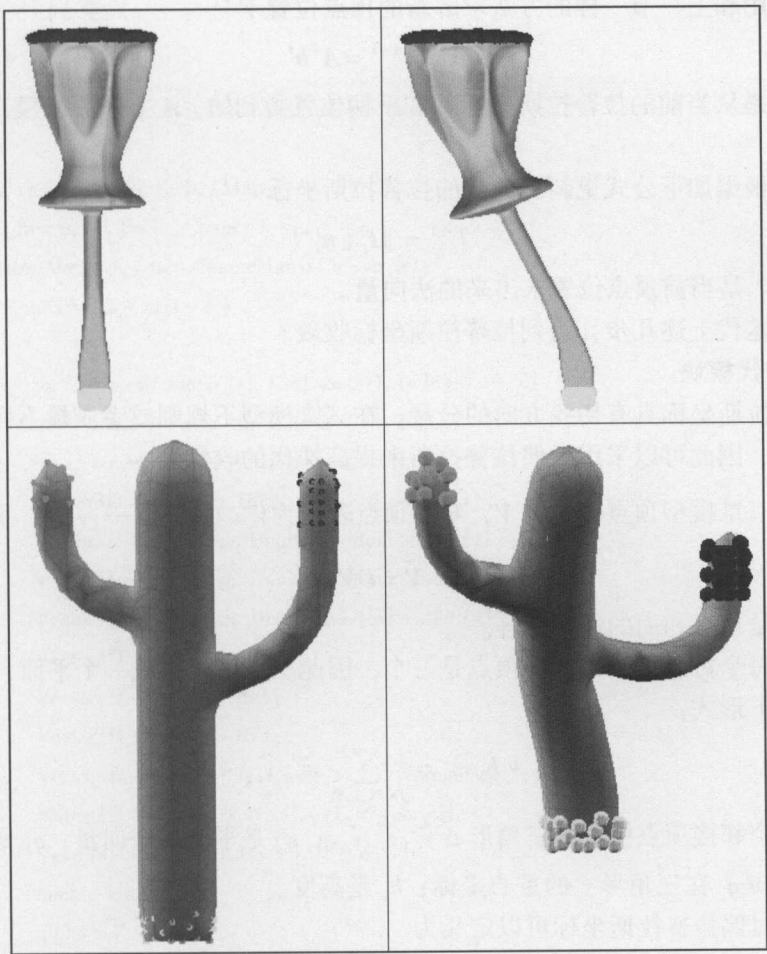


图 6-9 三维模型变形（续）

6.3.2 数学模型构建

拉普拉斯迭代算法的数学系统构建和上一节讲述的变形算法类似，但是需要在每次迭代的时候保持微分坐标向量的大小和更新这些向量的方向，这是两种变形算法的不同。另一个不同的地方在于迭代法采用了对偶模型。因为三位模型通常具有不规则形状的三角形面和连接关系，而对偶模型每个顶点只有三个邻居顶点，因此比较规则。从而在对偶模型上计算比较稳定。拉普拉斯迭代变形算法可以分为两个大的模块：迭代算法系统构建模块和在对偶模型上进行迭代操作的模块。

1. 三维模型上迭代模块

第一步：采用余切权重计算拉普拉斯坐标。

$$w_{ij} = \cot\alpha_i + \cot\beta_i$$

第二步：让 V^t 和 I^t 分别表示在第 t 步时的顶点位置和顶点的拉普拉斯坐标。

第三步：也就是 $V^0 = V, I^0 = I$

第四步：用和上一节一样的方法求解新的顶点位置 V^{t+1}

$$A^T A V^{t+1} = A^T b^t$$

其中， b^t 是从当前的拉普拉斯坐标 l^t 和手柄位置得到的， A 是从原来模型构建的，一直保持不变。

第五步：根据如下公式更新当前步的拉普拉斯坐标 l^{t+1} ：

$$l_i^{t+1} = \|l_i^0\| n_i^{t+1}$$

其中， n_i^{t+1} 是当前顶点位置求出来的法向量。

第六步：迭代上述几步，直到拉普拉斯坐标收敛。

2. 对偶迭代模块

因为拉普拉斯坐标具有切线方向的分量，在三维模型不规则或者质量不好的情况下，迭代法收敛很差。因此可以采用对偶拉普拉斯来提高迭代的收敛性。

第一步：三维模型顶点表示为 V ，对偶顶点表示为 $\tilde{V} = (\tilde{v}_1, \tilde{v}_2, \dots, \tilde{v}_{n_d})$ ，那么：

$$\tilde{V} = DV$$

其中， D 是顶点和面的邻接矩阵。

第二步：每个对偶顶点的邻接顶点是三个，因此三点可以构成一个平面，从而对偶顶点可以表示为如下形式：

$$\tilde{v}_i = \tilde{q}_i + h_i \tilde{n}_i = \sum_{j \in \{1,2,3\}} \tilde{w}_{i,j} \tilde{v}_{i,j} + h_i \tilde{n}_i$$

其中，三个邻接顶点构成的三角形 $\Delta \tilde{v}_{i,1} \tilde{v}_{i,2} \tilde{v}_{i,3}$ ； \tilde{n}_i 是平面的法向量； \tilde{q}_i 是顶点 \tilde{v}_i 在平面上的投影； $\tilde{w}_{i,j}$ 是 \tilde{q}_i 在三角形上的重心坐标； h_i 是高度。

第三步：对偶拉普拉斯坐标可以定义为

$$\tilde{I}_i = -h_i \tilde{n}_i = \sum_{j \in \{1,2,3\}} \tilde{w}_{i,j} (\tilde{v}_{i,j} - \tilde{v}_i)$$

矩阵形式表示为

$$\tilde{I}_i = \tilde{L}\tilde{V} = \tilde{L}DV$$

第四步：对偶拉普拉斯坐标向量和主拉普拉斯坐标向量不一样的地方是，对偶拉普拉斯向量没有切线方向的分量，总是在法向上。从而变形系统形式如下。

$$\arg \min_{V'} \|\tilde{L}DV' - \tilde{I}\|^2$$

第五步：相应的优化问题可以变为如下线性系统。

$$\tilde{A}^T \tilde{A} V' = \tilde{A}^T \tilde{b}$$

第六步：可以使用上述的迭代法方法求解。

6.3.3 迭代法核心代码

和上一节的代码相比，迭代算法的代码主要增加了构建对偶矩阵，以及相应的对偶拉普拉斯向量和迭代法求解线性系统等几个模块。其他的线性系统最小二乘法求解，更新顶点坐

标都和上一节的代码类似。

第一步：构建对偶矩阵。

```
public SparseMatrix BuildMatrixDual(TriMesh mesh)
{
    int vn = mesh.Vertices.Count;
    int fn = mesh.Faces.Count;
    SparseMatrix L = new SparseMatrix(fn, vn, 6);
    for(int i = 0; i < fn; i++)
    {
        int f1 = mesh.Faces[i].GetFace(0).Index;
        int f2 = mesh.Faces[i].GetFace(1).Index;
        int f3 = mesh.Faces[i].GetFace(2).Index;
        Vector3D dv = mesh.DualGetVertexPosition(i);
        Vector3D dv1 = mesh.DualGetVertexPosition(f1);
        Vector3D dv2 = mesh.DualGetVertexPosition(f2);
        Vector3D dv3 = mesh.DualGetVertexPosition(f3);
        Vector3D u = dv - dv3;
        Vector3D v1 = dv1 - dv3;
        Vector3D v2 = dv2 - dv3;
        Vector3D normal = (v1.Cross(v2)).Normalize();
        Matrix3D M = new Matrix3D(v1, v2, normal);
        Vector3D coord = M.Inverse() * u;
        double alpha;
        alpha = 1.0/3.0;
        L.AddValueTo(i, mesh.Faces[i].GetVertex(0).Index, alpha);
        L.AddValueTo(i, mesh.Faces[i].GetVertex(1).Index, alpha);
        L.AddValueTo(i, mesh.Faces[i].GetVertex(2).Index, alpha);
        alpha = coord[0]/3.0;
        L.AddValueTo(i, mesh.Faces[f1].GetVertex(0).Index, -alpha);
        L.AddValueTo(i, mesh.Faces[f1].GetVertex(1).Index, -alpha);
        L.AddValueTo(i, mesh.Faces[f1].GetVertex(2).Index, -alpha);
        alpha = coord[1]/3.0;
        L.AddValueTo(i, mesh.Faces[f2].GetVertex(0).Index, -alpha);
        L.AddValueTo(i, mesh.Faces[f2].GetVertex(1).Index, -alpha);
        L.AddValueTo(i, mesh.Faces[f2].GetVertex(2).Index, -alpha);
        alpha = (1.0 - coord[0] - coord[1])/3.0;
        L.AddValueTo(i, mesh.Faces[f3].GetVertex(0).Index, -alpha);
        L.AddValueTo(i, mesh.Faces[f3].GetVertex(1).Index, -alpha);
        L.AddValueTo(i, mesh.Faces[f3].GetVertex(2).Index, -alpha);
    }
}
```

```
L.SortElement();
return L;
}
```

第二步：构建对偶拉普拉斯。

```
public double[,] ComputeLaplacianDual(TriMesh mesh)
{
    int fn = mesh.Faces.Count;
    double[,] laplacian = new double[3][];
    laplacian[0] = new double[fn];
    laplacian[1] = new double[fn];
    laplacian[2] = new double[fn];
    for(int i = 0; i < fn; i++)
    {
        int f1 = mesh.Faces[i].GetFace(0).Index;
        int f2 = mesh.Faces[i].GetFace(1).Index;
        int f3 = mesh.Faces[i].GetFace(2).Index;
        Vector3D u = mesh.DualGetVertexPosition(i);
        Vector3D v1 = mesh.DualGetVertexPosition(f1);
        Vector3D v2 = mesh.DualGetVertexPosition(f2);
        Vector3D v3 = mesh.DualGetVertexPosition(f3);
        Vector3D normal = ((v1 - v3).Cross(v2 - v3)).Normalize();
        Matrix3D m = new Matrix3D(v1 - v3, v2 - v3, normal);
        Vector3D coord = m.Inverse() * (u - v3);
        laplacian[0][i] = normal.x * coord[2];
        laplacian[1][i] = normal.y * coord[2];
        laplacian[2][i] = normal.z * coord[2];
    }
    return laplacian;
}
```

第三步：更新拉普拉斯坐标。

```
public void UpdatelapCoordinate()
{
    for(int i = 0; i < mesh.Faces.Count; i++)
    {
        int f1 = mesh.Faces[i].GetFace(0).Index;
        int f2 = mesh.Faces[i].GetFace(1).Index;
        int f3 = mesh.Faces[i].GetFace(2).Index;
        Vector3D p1 = mesh.DualGetVertexPosition(f1);
        Vector3D p2 = mesh.DualGetVertexPosition(f2);
```



```

Vector3D p3 = mesh. DualGetVertexPosition(f3);
Vector3D normal = ((p1 - p3). Cross(p2 - p3)). Normalize();
b[0][i] = normal. x * h[i];
b[1][i] = normal. y * h[i];
b[2][i] = normal. z * h[i];
}
}

```



6.4 ARAP 变形算法

6.4.1 算法思想

三维模型的形状 (Shape) 指的是不随模型的位置 (Position) 和角度 (Orientation) 发生变化的属性。保持形状不变就是指模型只发生旋转 (Rotation) 和位移 (Translation) 的变化, 而不发生拉伸 (Scale) 和扭曲 (Shear) 的变化。在三维模型变形的操作中, 为了满足特定的限制, 如控制点的位置, 那么拉伸和扭曲是不可避免的。因此在三维模型变形操作中, 需要的不是整个三维模型整体上的不发生拉伸和扭曲, 而是局部上尽可能少的拉伸和扭曲。也就是当三维模型在局部上的变形尽可能的没有拉伸和扭曲, 那么全局上的细节就可以被保持, 这种变形算法称为 ASAP (As Rigid as Possible) 算法。

ARAP 算法的核心思想是把三维模型的表面由若干互相重合的小的局部网格所覆盖。这个局部网格可以是每个顶点及这个顶点所相邻的面构成。那么对于每个顶点都有一个局部网格, 这些网格互相重叠, 并且能够覆盖整个三维模型。ARAP 算法在变形的过程中, 保持每个顶点的局部网格尽可能不发生拉伸和扭曲。这种算法里局部只能保持尽可能的减少拉伸和扭曲, 而不能完全没有拉伸和扭曲, 因为假如局部完全没有拉伸扭曲, 那么整体上也就不发生任何变形。对于变形手柄平移、旋转所带来的限制条件, 可以平均的光滑地分配到所有的局部小网格, 从而在整体上尽可能保持原来的细节不变。也就是不会在三维模型的某个地方出现非常明显的扭曲, 因为扭曲被分配到每个小网格, 在这样的思路下, ARAP 算法所构建的数学系统是一个非线性系统, 可以通过迭代的算法把这个非线性系统转换为线性的系统。也就是虽然整体上来说, 系统是非线性的, 但是迭代的每一步都是线性的。

相比于前两种算法, ASAP 算法是拉普拉斯框架下变形效果最好的算法。在手柄的限制条件下, ASAP 算法也会尽可能保持边长不变。因为 ASAP 算法是基于能量函数设计的, 每一步迭代都会减少能量, 从而可以保证迭代的收敛。ASAP 算法很好地处理了拉普拉斯坐标的旋转问题, 从而可以支持手柄位移产生的旋转和把手柄的旋转插值到其他顶点上。图 6-10 是仙人掌和狗的三维模型用 ARAP 变形算法变形的效果图。从狗尾巴的变形可以看出来, 尾巴尖把柄的平移带来的旋转可以平滑地传递到整个狗尾巴, 从而生成比较光滑自然的变形效果。

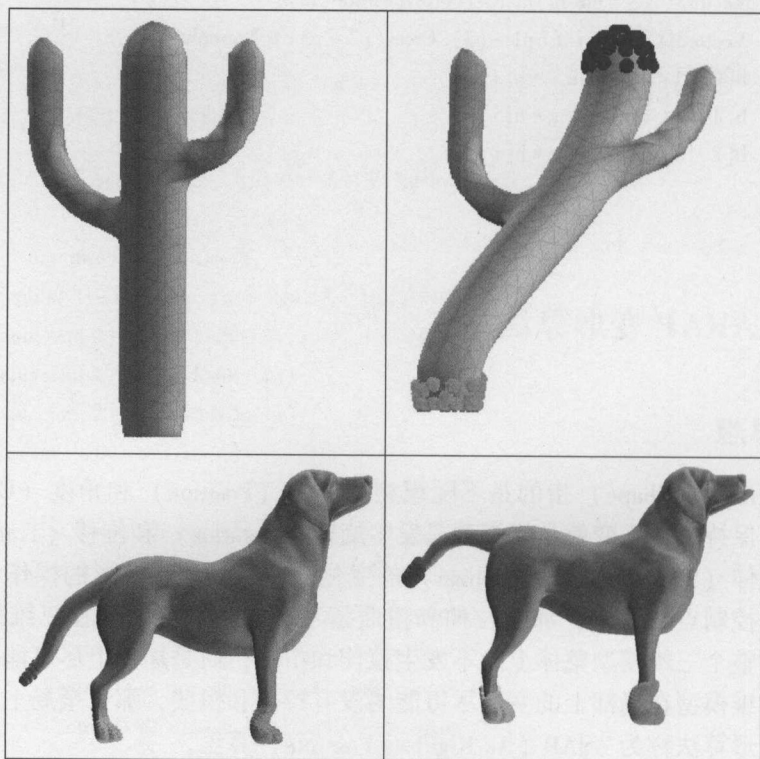


图 6-10 ARAP 算法变形效果

6.4.2 数学模型构建

ARAP 变形算法数学模型基于上诉的算法思路，对于局部上的每个顶点需要构建相应的矩阵来表示这个顶点周围边的旋转。这个矩阵是刚性的，这样才能保证局部不发生拉升和扭曲。但是由于得到的矩阵并不一定是刚性的，因此需要进行分解，得到其中刚性的部分。得到单个网格的数学模型之后，就可以进一步把所有网格求和，从而可以得到整个三维模型进行 ARAP 变形的数学模型。

1. 单个局部网格变形

第一步：让 c_i 表示对应于顶点 i 的局部网格，也就是此顶点相邻的所有的面。

第二步： c_i 变形后表示为 c'_i ，假如变形 $c \rightarrow c'$ 是刚性的（Rigid），也就是没有拉伸和扭曲，那么有如下公式。

$$p'_i - p'_j = R_i(p_i - p_j), \forall j \in N(i)$$

其中， R_i 是一个旋转矩阵。

第三步：假如变形不是刚性的，那么可以定义如下一个能量函数，求得的旋转矩阵使此能量函数最小。

$$E(c_i, c'_i) = \sum_{j \in N(i)} w_{ij} \| (p'_i - p'_j) - R_i(p_i - p_j) \|^2$$

第四步：假设边长为 $e_{ij} = p_i - p_j$ ，从而可以得到新的能量函数。

$$\begin{aligned}
& \sum_j w_{ij} (\mathbf{e}'_{ij} - \mathbf{R}_i \mathbf{e}_{ij})^T (\mathbf{e}'_{ij} - \mathbf{R}_i \mathbf{e}_{ij}) \\
&= \sum_j w_{ij} (\mathbf{e}'_{ij}{}^T \mathbf{e}'_{ij} - 2\mathbf{e}'_{ij}{}^T \mathbf{R}_i \mathbf{e}_{ij} + \mathbf{e}_{ij}{}^T \mathbf{R}_i^T \mathbf{R}_i \mathbf{e}_{ij}) \\
&= \sum_j w_{ij} (\mathbf{e}'_{ij}{}^T \mathbf{e}'_{ij} - 2\mathbf{e}'_{ij}{}^T \mathbf{R}_i \mathbf{e}_{ij} + \mathbf{e}_{ij}{}^T \mathbf{e}_{ij})
\end{aligned}$$

第五步：上述推导中和旋转矩阵无关的项可以省略，从而得到

$$\begin{aligned}
\arg \min \sum_j -2w_{ij} \mathbf{e}'_{ij}{}^T \mathbf{R}_i \mathbf{e}_{ij} &= \arg \max_{\mathbf{R}_i} \sum_j w_{ij} \mathbf{e}'_{ij}{}^T \mathbf{R}_i \mathbf{e}_{ij} \\
&= \arg \max_{\mathbf{R}_i} \text{Tr} \left(\sum_j w_{ij} \mathbf{R}_i \mathbf{e}_{ij} \mathbf{e}'_{ij}{}^T \right) \\
&= \arg \max_{\mathbf{R}_i} \text{Tr} \left(\mathbf{R}_i \sum_j w_{ij} \mathbf{e}_{ij} \mathbf{e}'_{ij}{}^T \right)
\end{aligned}$$

第六步：假设 \mathbf{D}_i 是一个对角矩阵，包含权重 w_{ij} ， \mathbf{P}_i 是包含 \mathbf{e}_{ij} 的 $3 \times |N(v_i)|$ 矩阵，那么

$$\mathbf{S}_i = \sum_{j \in N(i)} w_{ij} \mathbf{e}_{ij} \mathbf{e}'_{ij}{}^T = \mathbf{P}_i \mathbf{D}_i \mathbf{P}_i^T$$

第七步：对于第五步推导的能量函数来说，当 $\mathbf{R}_i \mathbf{S}_i$ 是半正定的矩阵，那么能量函数得到最小值。

第八步：旋转矩阵可以从 $\mathbf{S}_i = \mathbf{U}_i \Sigma_i \mathbf{V}_i^T$ 分解来得到，也就是 $\mathbf{R}_i = \mathbf{V}_i \mathbf{U}_i^T$

2. 整个三维模型的变形

第一步：整个三维模型上的能量函数可以由所有单个网格上的能量函数之和得到：

$$\begin{aligned}
E(S') &= \sum_{i=1}^n w_i E(c_i, c'_i) \\
&= \sum_{i=1}^n w_i \sum_{j \in N(i)} w_{ij} \| (p'_i - p'_j) - \mathbf{R}_i (p_i - p_j) \|^2
\end{aligned}$$

其中， w_i 、 w_{ij} 是分别定义在面上和边上的权重。

第二步：这个能量函数依赖于三维模型变形前和变形后的顶点坐标值，但是变形前的坐标值是固定的，因此只依赖于变形后的坐标值。

第三步：边上的权重可以采取下面余切权重，面上的权重可以是 $w_i = 1$ 或者顶点对应的面积 A_i 。

$$w_{ij} = \frac{1}{2} (\cot \alpha_{ij} + \cot \beta_{ij})$$

第四步：能量函数的最小值可以通过对变形后顶点的位置求导得到。

$$\begin{aligned}
\frac{\partial E(S')}{\partial p'_i} &= \frac{\partial}{\partial p'_i} \left(\sum_{j \in N(i)} w_{ij} \| (p'_i - p'_j) - \mathbf{R}_i (p_i - p_j) \|^2 + \sum_{j \in N(i)} w_{ji} \| (p'_j - p'_i) - \mathbf{R}_j (p_j - p_i) \|^2 \right) \\
&= \sum_{j \in N(i)} 2w_{ij} ((p'_i - p'_j) - \mathbf{R}_i (p_i - p_j)) + \sum_{j \in N(i)} -2w_{ji} ((p'_j - p'_i) - \mathbf{R}_j (p_j - p_i))
\end{aligned}$$

第五步：根据 $w_{ij} = w_{ji}$ ，从而得到：

$$\frac{\partial E(S')}{\partial p'_i} = \sum_{j \in N(i)} 4w_{ij} \left((p'_i - p'_j) - \frac{1}{2} (\mathbf{R}_i + \mathbf{R}_j) (p_i - p_j) \right)$$

第六步：把导数设置为 0，得到如下方程。

$$\sum_{j \in N(i)} w_{ij} (p'_i - p'_j) = \sum_{j \in N(i)} \frac{w_{ij}}{2} (\mathbf{R}_i + \mathbf{R}_j) (p_i - p_j)$$

第七步：上述方程左边是拉普拉斯矩阵，从而方程组的矩阵形式为

$$Lp' = b$$

第八步：假如手柄位置的限制条件，构成完成的系统为

$$p'_j = c_k$$

第九步：上述系统中，虽然未知数是变形后顶点的位置，但是旋转矩阵 R_i 也依赖于变形后顶点的位置才能得到，因此这是一个非线性的问题。

第十步：可以通过迭代的方法，把非线性的问题变为线性的问题。

第十一步：先得到一个变形后顶点位置的初始值 p'_0 ，然后计算旋转矩阵。

第十二步：得到旋转矩阵后，再用第六步的方程组计算变形后的顶点位置。

第十三步：迭代进行上两步，直到收敛。

6.4.3 ARAP 核心代码

虽然 ARAP 算法的思路和数学建模是基于局部网格的，但是最终推导出来的也是一个和上两节变形算法类似的线性系统，依赖于拉普拉斯矩阵。ARAP 的代码和之前两个变形算法的结构基本一致，但是需要求解旋转矩阵和对旋转矩阵进行分解，以及分解后更新方程组右边的向量等模块。

第一步：构建拉普拉斯矩阵。

```
public SparseMatrix BuildMatrixRigid( TriMesh mesh)
{
    int n = mesh. Vertices. Count;
    SparseMatrix L = new SparseMatrix( n, n );
    for( int i = 0; i < mesh. Faces. Count; i ++ )
    {
        int c1 = mesh. Faces[ i ]. GetVertex( 0 ). Index;
        int c2 = mesh. Faces[ i ]. GetVertex( 1 ). Index;
        int c3 = mesh. Faces[ i ]. GetVertex( 2 ). Index;
        Vector3D v1 = mesh. Faces[ i ]. GetVertex( 0 ). Traits. Position;
        Vector3D v2 = mesh. Faces[ i ]. GetVertex( 1 ). Traits. Position;
        Vector3D v3 = mesh. Faces[ i ]. GetVertex( 2 ). Traits. Position;
        double cot1 = ( v2 - v1 ). Dot( v3 - v1 ) / ( v2 - v1 ). Cross( v3 - v1 ). Length();
        double cot2 = ( v3 - v2 ). Dot( v1 - v2 ) / ( v3 - v2 ). Cross( v1 - v2 ). Length();
        double cot3 = ( v1 - v3 ). Dot( v2 - v3 ) / ( v1 - v3 ). Cross( v2 - v3 ). Length();
        L. AddValueTo( c1, c2, - cot3 / 2 ); L. AddValueTo( c2, c1, - cot3 / 2 );
        L. AddValueTo( c2, c3, - cot1 / 2 ); L. AddValueTo( c3, c2, - cot1 / 2 );
        L. AddValueTo( c3, c1, - cot2 / 2 ); L. AddValueTo( c1, c3, - cot2 / 2 );
    }
    for( int i = 0; i < n; i ++ )
    {
        double sum = 0;
        foreach( SparseMatrix. Element e in L. Rows[ i ] )
```

```

    {
        sum += e. value;
    }
    L. AddValueTo(i,i, - sum);
}
L. SortElement();
return L;
}

```

第二步：计算旋转矩阵。

```

public void ComputerRigidRotation()
{
    int n = mesh. Vertices. Count;
    for(int i = 0 ;i < n;i ++ )
    {
        Matrix3D S = new Matrix3D();
        Vector3D u1 = mesh. Vertices[i]. Traits. Position ;
        Vector3D u1plus = backUpPosition[i];
        foreach( TriMesh. Vertex neighbor in mesh. Vertices[i]. Vertices)
        {
            Vector3D u21 = u1 - neighbor. Traits. Position;
            Vector3D u21plus = u1plus - backUpPosition[ neighbor. Index];
            S += u21plus. OuterCross( u21 ) * - L. FindElement(i,neighbor. Index). value;
        }
        rigidRotation[i] = S. SVDRotation();
        double value = rigidRotation[i]. Det();
        if( value < 0)
            rigidRotation[i] = S. SVDRotationMinus();
    }
}

```

第三步：用得到的旋转矩阵计算方程组的右边向量。

```

public void ComputeRigidB()
{
    for(int i = 0;i < mesh. Vertices. Count;i ++ )
    {
        b[0][i] = 0;
        b[1][i] = 0;
        b[2][i] = 0;
        Vector3D u1plus = backUpPosition[i];
        foreach( TriMesh. Vertex neighbor in mesh. Vertices[i]. Vertices)
        {

```

```

        Vector3D u21plus = u1plus - backUpPosition[ neighbor. Index ];
        Vector3D vectorB = ( rigidRotation[ i ] + rigidRotation[ neighbor. Index ])
            /2 * u21plus * - L. FindElement( i, neighbor. Index ). value;
        b[ 0 ][ i ] += vectorB. x;
        b[ 1 ][ i ] += vectorB. y;
        b[ 2 ][ i ] += vectorB. z;
    }
}

```

第四步：进行迭代。

```

public override void Deform()
{
    base. Deform();
    for( int i = 0; i < 10; i ++ )
    {
        ComputerRigidRotation();
        ComputeRigidB();
        ComputeDeform();
    }
}

```


第7章

拉普拉斯模型处理算法



7.1 三维模型近似算法

7.1.1 三维模型近似概述

拉普拉斯矩阵是三维模型处理中一个重要的概念和工具。不仅仅可以用在三维模型的变形上,还可以用在三维模型的近似、光滑、优化、提取骨骼、计算最短距离等各种各样的操作上。所有三维模型上的处理操作都可以看作三维模型的改变,而对三维模型的改变归根结底都是要改变三维模型顶点的位置,从而使三维模型满足特定的需求和条件。只有三维模型顶点的位置改变后,三维模型才会发生变化。三维模型顶点的位置可以看作一个向量,那么改变后的位置是另一个向量。也就是所有三维模型上的操作都是输入一个向量、输出一个向量。针对特定的处理操作,需要构建一个数学模型,通过这个数学模型,把输入的向量变为输出的向量。大部分三维模型的操作都可以构建为一个线性系统的数学模型,这些线性的核心就是拉普拉斯矩阵。

对于一个三维模型来说,通常有几千几万个点,数据量非常大。这样大的模型在网络传递,内存加载时都比较耗费时间。有时候需要把这些模型进行处理得到一个数据量比较小,但是和原来的三维模型在形状上比较近似的三维模型。三维模型近似算法指的是采用更少的信息来得到和原来模型形似的模型的算法。

一个三维模型由拓扑信息和几何信息构成。几何信息也就是三维模型顶点的位置。如果三维模型没有几何信息,那么这个三维模型可以称为是一个连接模型(Connectivity Mesh)。一个连接模型没有形状,只有顶点之间的连接关系。一个连接模型可以通过任意一个三维模型移除几何信息后得到。

如果已经知道连接模型的拓扑信息,还确定三维模型若干个顶点的几何信息,那么这个三维模型其他顶点的几何信息可以通过拓扑信息进行计算得到。也就是通过拓扑信息和一部分几何信息,可以得到近似三维模型的形状。从而如果给定一个三维模型,可以移除这个三维模型的几何信息,只保留拓扑连接信息和若干个顶点,然后通过这个方法得到近似于原来模型的三维模型。随着已知顶点的增多,求解出来的模型会越来越近似于原来的三维模型。这种方法求得的模型因为是用最小二乘法得到,因此也叫最小二乘法模型(Least Square Mesh)。近似模型的效果如图7-1所示。其中图左是原始模型,图右是近似模型,图右的顶点是已知的固定顶点。

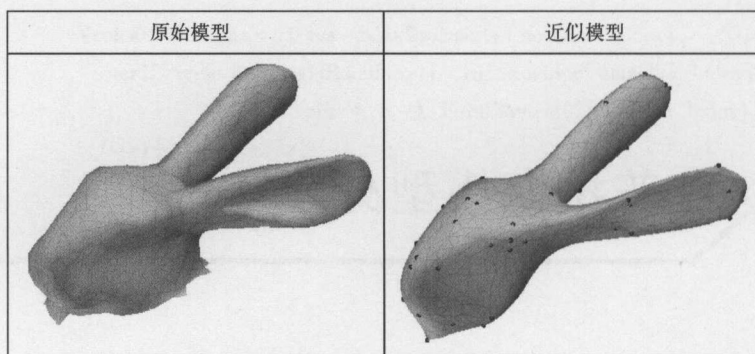


图 7-1 近似模型

7.1.2 数学系统构建和代码

1. 数学系统构建过程

在上一章拉普拉斯变形算法的介绍中可知，通过三维模型的拉普拉斯变换，可以得到微分坐标。微分坐标表示的是三维模型的细节。而近似模型就是把三维模型的细节去掉得到的模型。在去掉三维模型细节的同时，还要保持三维模型的整体形状。但是假如三维模型所有细节都消失，整体形状就难以保持。因此需要固定若干个不同部位的顶点，从而用这些顶点来约束三维模型的整体形状。近似算法数学模型的构建首先需要得到三维模型的拉普拉斯矩阵，其次需要确定固定的顶点，把这些顶点作为数学系统的约束条件。最终通过最小二乘法来得到尽可能满足约束条件的三维模型每个顶点的位置。

第一步：通过三维模型可以得到拉普拉斯矩阵

$$L_{ij} = \begin{cases} 1 & i=j \\ -\frac{1}{d_i} & (i,j) \in E \\ 0 & \text{其他} \end{cases}$$

第二步：为 x 、 y 、 z 坐标分别构建如下线性系统。

$$Lx=0, Ly=0, Lz=0$$

第三步：给定 m 个控制顶点，也就是已知顶点的位置，

$$C = \{s_1, s_2, \dots, s_m\}$$

第四步：其中每个顶点为

$$V_s = (x_s, y_s, z_s), s \in C$$

第五步：然后把这 m 个顶点加入第二步构建的线性方程，从而得到如下系统。

$$Ax = b$$

第六步：这是一个长方形 $(n+m) \times n$ 的系统。

其中，

$$A = \begin{pmatrix} L \\ F \end{pmatrix}, F_{ij} = \begin{cases} 1 & j = s_i \in C \\ 0 & \text{其他} \end{cases}$$

$$b_k = \begin{cases} 0 & k \leq n \\ x_{s_{k-n}} & n < k \leq n+m \end{cases}$$

第七步：这个系统可以用最小二乘法求解，也就是求解未知数满足如下最小值。

$$\|Ax - b\|^2 = \|Lx\|^2 + \sum_{s \in C} |x_s - v_s^{(x)}|^2$$

第八步：最小值为

$$x = (A^T A)^{-1} A^T b$$

例如，如图 7-2 所示，在一个简单的网格模型中，如果 V_1 和 V_4 的顶点位置已知，其他的顶点位置未知，那么构建的矩阵 A 如图 7-2 所示。

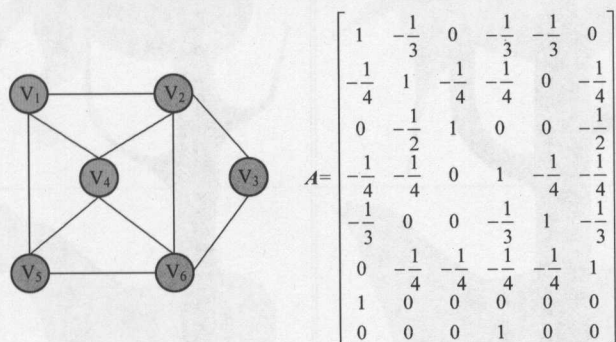


图 7-2 图形和矩阵

这个系统和变形系统的区别是，方程的右边 b 的上部不是拉普拉斯坐标，而是 0。因为假如拉普拉斯坐标已知，那么所有三维模型的顶点位置就确定了。

2. 近似模型核心代码函数

近似算法步骤和前一章节变形算法的整体流程都一样，大部分函数可以参考上一章节的代码。只是在构建方程组右边向量的时候，向量的上部分值为 0，而在变形算法中，向量的上部分值为微分坐标数值。

```
public override double[ ][ ] BuildLaplacian()
{
    double[ ][ ] lap = new double[3][ ];
    for( int i=0; i < 3; i++ )
    {
        lap[i] = new double[ mesh.Vertices.Count ];
        for( int j=0; j < mesh.Vertices.Count; j++ )
        {
            lap[i][j] = 0;
        }
    }
    return lap;
}
```

7.1.3 近似模型算法效果图

近似模型输入的信息是原来模型的拉普拉斯矩阵和若干的固定顶点，输出的信息是新的三维模型。这些固定顶点应该均匀地分布在三维模型的表面上，这样才能得到更好的效果。假如对于一个狗的三维模型来说，所有顶点都分布在头部，那么狗的身体和尾巴等其他部位

就很难重建出来。同时选择的点越多，得到的模型和原来的模型越相似。假如全部的点都固定，那么就与原来模型一样了。

(1) 在图 7-3 中，根据选择的已知控制点的不同，重建后的效果不同。选择的点越多，近似的效果越好。

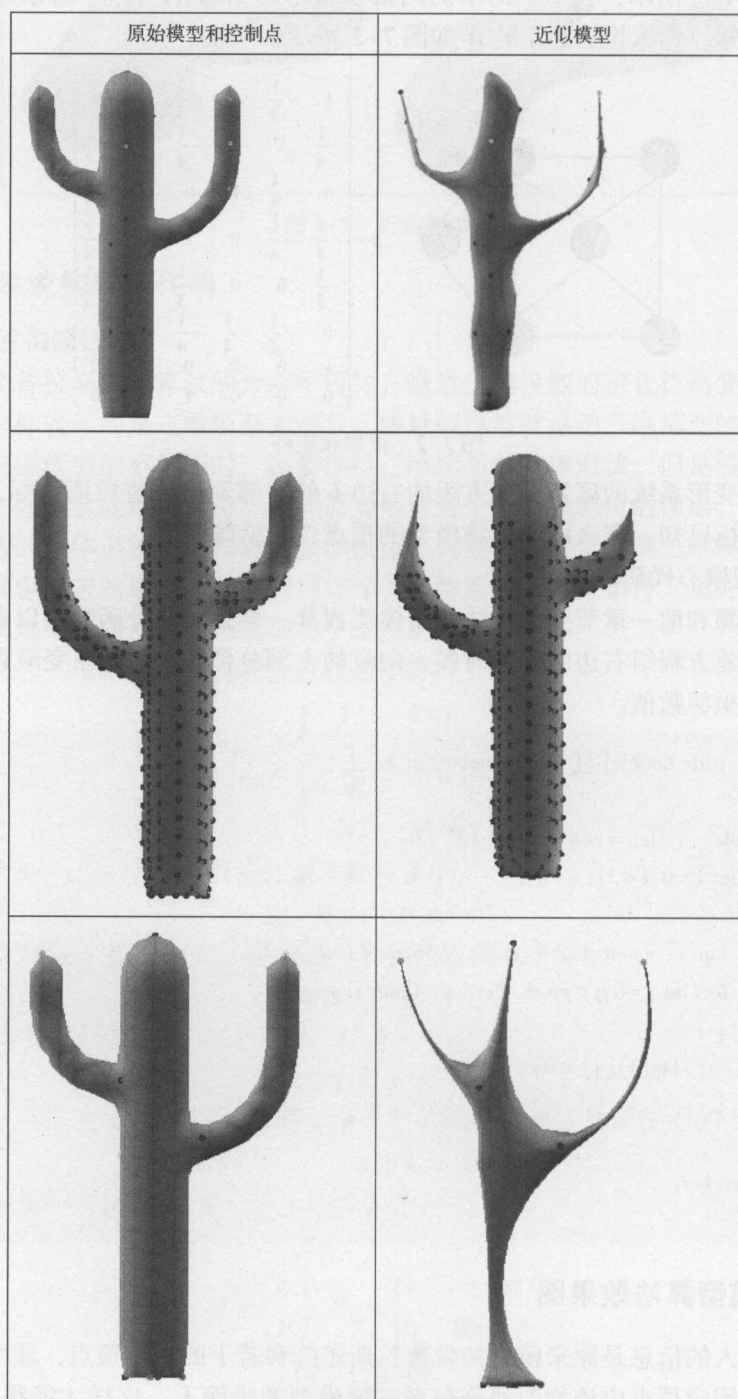


图 7-3 不同控制点得到的近似模型

(2) 其实拓扑信息本身也包含有一些几何信息,如图7-4,虽然狗的头部和马的头部顶点至于拓扑信息,在重新构建之后,仍旧能还原一点原来的几何信息。

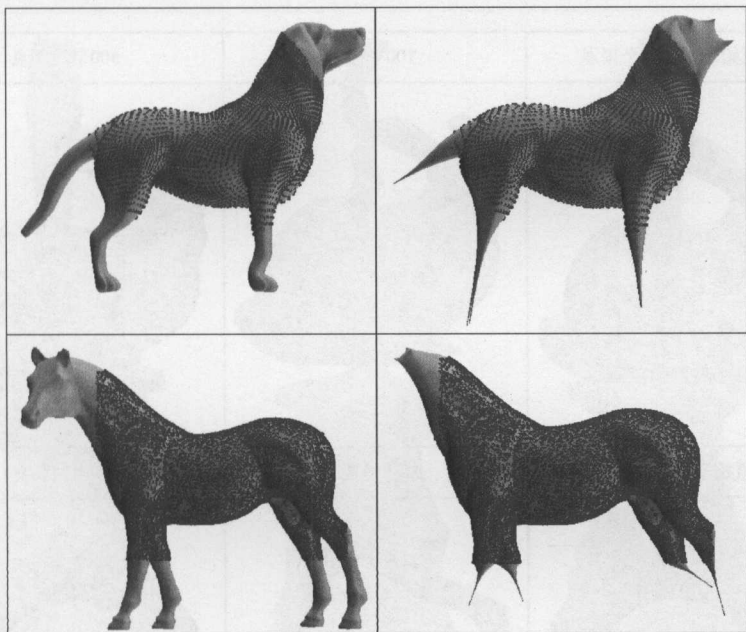


图 7-4 几何信息恢复

(3) 图7-5中,骆驼的驼峰部分被删除,那么通过重建方法可以得到一个没有驼峰的骆驼。

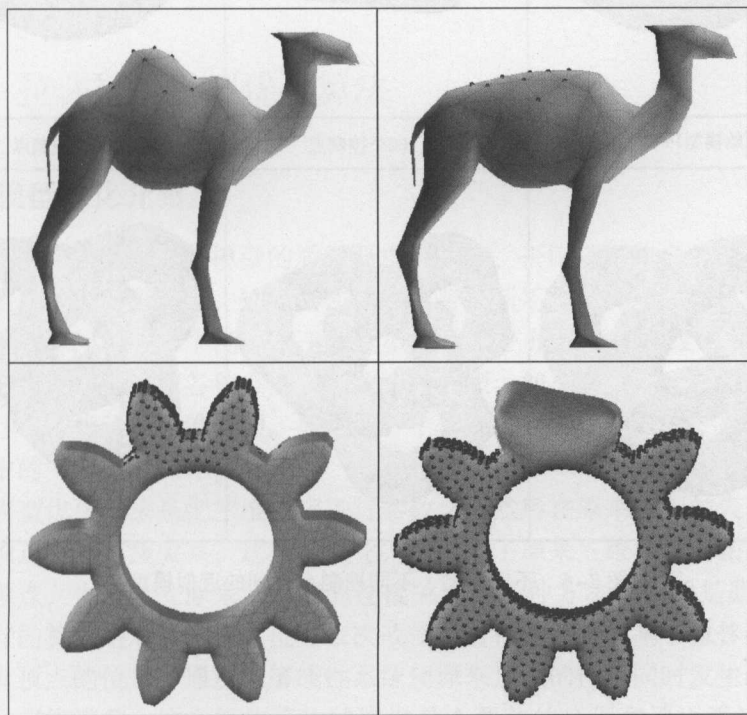


图 7-5 用近似方法进行模型光滑

（4）图 7-6 是随机选择若干控制点来重建原始模型，从中可以看出不到原来顶点数量的十分之一的顶点就可以很好地重建出原来的模型。

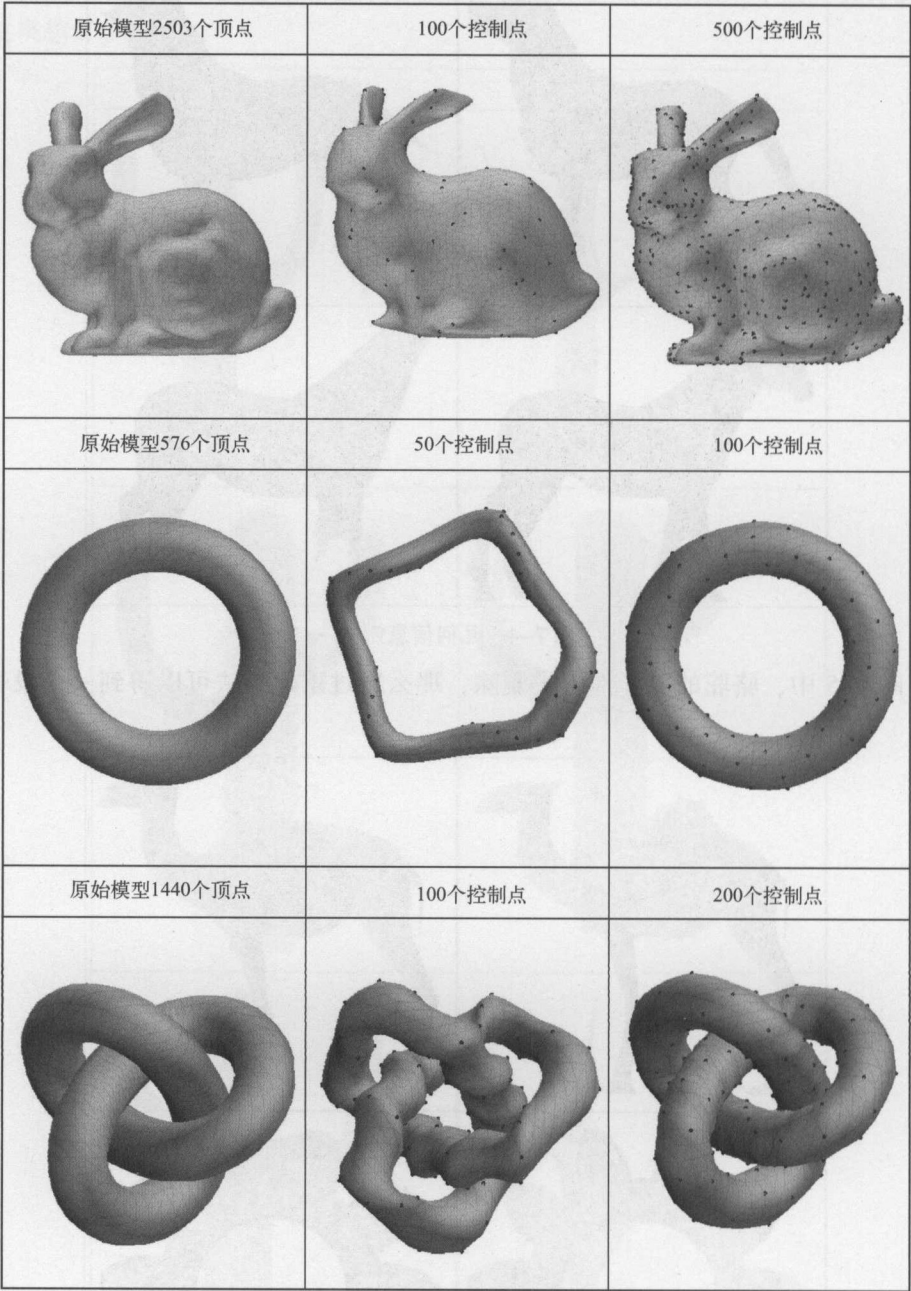


图 7-6 不同模型、不同控制点得到的近似模型

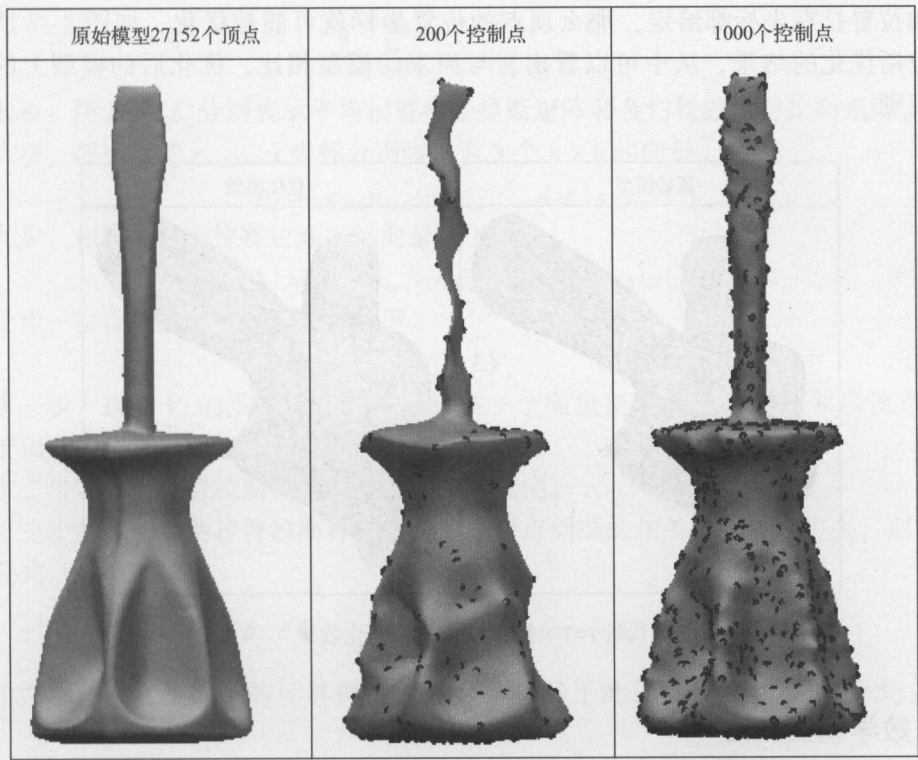


图 7-6 不同模型、不同控制点得到的近似模型（续）



7.2 拉普拉斯模型优化算法

7.2.1 三维模型优化介绍

很多三维模型的操作对三维模型的质量要求比较高，否则就可能会失败。例如，在三维模型中有些三角形的角度是钝角，这样会在计算拉普拉斯矩阵的时候得到负值，从而在后续的解线性方程时出现问题。一个优化的三维模型是每个三角形面的角度都是 60 度，三角形的面积大小也一样。但是通过各种方法得到的三维模型，如三维扫描，或者用软件制作的三维模型往往不是一个优化的三维模型。如何把一个非优化的三角形变为一个优化的三角形是三维模型处理中的一个重要操作。

一类三维模型优化算法是把三维模型进行参数化，然后在原来模型的表面上重新采样顶点和构建这些顶点新的连接关系，这样的优化方法改变了原来三维模型的拓扑。还有一类方法不生成新的顶点，也不改变原来顶点间的连接关系，这种方法改变的是原来顶点的位置。这类方法里面有的算法是对每个顶点位置依次改变，进行优化，而基于拉普拉斯的优化方法是同时改变所有顶点的位置，通过求解线性系统使原来顶点的位置同时发生改变，从而得到一个较为优化的三维模型。这个优化算法的启发是上两节的变形和逼近算法，也就是给定

若干顶点的几何位置和拉普拉斯位置，可以得到每个顶点新的坐标，如果所有的顶点位置坐标和拉普拉斯坐标都给定，那么顶点的位置坐标就可能被优化。如图 7-7 所示是基于拉普拉斯优化的结果，从中可以看出，与原来的模型相比，优化后的模型上的三角形面更加规则。

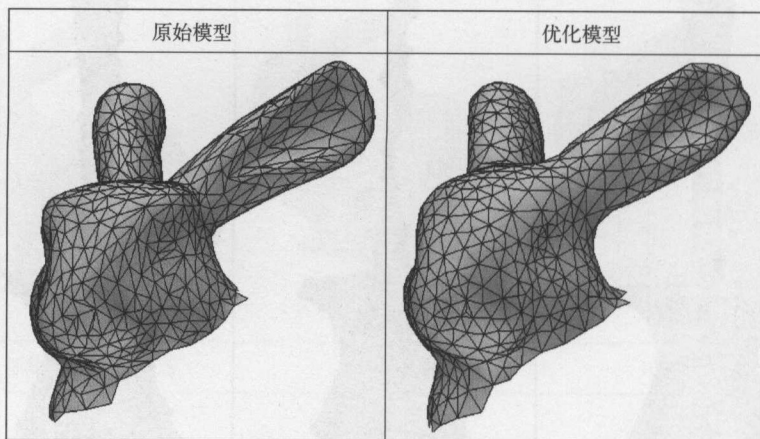


图 7-7 拉普拉斯模型优化效果

7.2.2 数学模型构建

对于一个三维模型来说，它的拉普拉斯算子或者拉普拉斯操作符有若干种。其中两大类是基于拓扑和几何信息的。拉普拉斯矩阵可以把顶点的坐标变为微分坐标，根据采用的拉普拉斯矩阵的不同，得到的微分坐标也不一样。基于拉普拉斯的三维模型优化算法就是使用两种不同的拉普拉斯矩阵。针对两种不同的拉普拉斯矩阵的特点，可以在构建一个能满足既能保持三维模型的形状和细节，又能够使三维模型的顶点发生改变，从而得到一个更规则的三角形。在这两个条件下，进行平衡，就能够实现三维模型优化的目标。

第一步：微分坐标的计算方式如下所示。

$$\delta_i = \sum_{\{i,j\} \in E} w_{ij}(v_j - v_i) = \left[\sum_{\{i,j\} \in E} w_{ij}v_j \right] - v_i$$

第二步：在本算法中，权重满足如下条件。

$$\sum_{\{i,j\} \in E} w_{ij} = 1$$

第三步：设定权重如下形式：

$$w_{ij} = \frac{w_{ij}}{\sum_{\{i,k\} \in E} w_{ik}}$$

第四步：根据拉普拉斯矩阵的不同，有两种选择。

$$w_{ij} = 1$$

$$w_{ij} = \cot\alpha + \cot\beta$$

第五步：第四步的第一个是平均权重，第二个是余切权重。

第六步：拉普拉斯矩阵大小 $n \times n$ ，形式如下所示。

$$L_{ij} = \begin{cases} -1 & i=j \\ w_{ij} & (i,j) \in E \\ 0 & \text{其他} \end{cases}$$

第七步：用 L_u 和 L_c 分别表示平均权重拉普拉斯矩阵和余切权重拉普拉斯矩阵。

第八步：把矩阵的 x 、 y 、 z 坐标分别表示为 3 个 $n \times 1$ 的向量。

$$V_d = [v_{1d}, v_{2d}, \dots, v_{nd}]^T, d \in \{x, y, z\}$$

第九步：相应的 3 个拉普拉斯坐标向量如下。

$$\Delta_d = [\delta_{1d}, \delta_{2d}, \dots, \delta_{nd}]^T, d \in \{x, y, z\}$$

第十步：拉普拉斯坐标计算公式如下。

$$\Delta_d = LV_d$$

第十一步：顶点 V_i 的平均拉普拉斯坐标是一个向量指向此顶点一层邻居顶点的中心，不依赖于顶点的位置。

第十二步：余切拉普拉斯坐标近似于顶点的法向。

第十三步：假如余切拉普拉斯计算时把每个顶点周围的相关面积加入进去，那么可以得到如下公式。

$$\bar{\kappa}_i n_i = \delta_i, c \bar{\kappa} = \frac{1}{4A(v_i)} \sum_{(i,j) \in E} (\cot \alpha + \cot \beta) (v_j - v_i)$$

第十四步：也就是这样的计算得到的微分坐标等于此顶点的法向 n_i 和平均曲率 $\bar{\kappa}_i$ 的乘积。

第十五步：用 $L_{c\bar{\kappa}}$ 表示上两步所示的拉普拉斯矩阵。

第十六步：在图 7-8 (a) 中，红色箭头表示平均拉普拉斯坐标向量，绿色箭头表示余切拉普拉斯坐标向量；在图 7-8 (b) 中，表示的是顶点相关的面积；图 7-8 (c) 右表示如果把顶点和周围的顶点位于同一个平面，那么余切拉普拉斯坐标小时，而平均拉普拉斯坐标不消失。

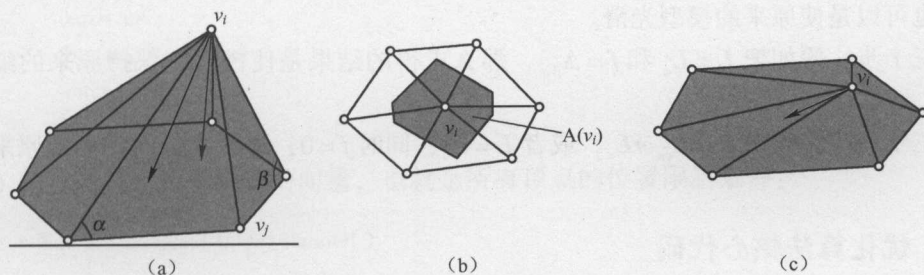


图 7-8 两种拉普拉斯坐标对比

第十七步：根据上一节的介绍，如果只知道平均拉普拉斯矩阵和若干个顶点的几何位置，那么其他顶点位置可以通过最小化如下的能量函数求出。

$$\|L_u V'_d\|^2 + \sum_{s \in C} w_s^2 |v'_{sd} - v_{sd}|^2$$

第十八步：从上一节可以知道，上一步的矩阵形式如下。

$$AV'_d = b$$

第十九步：其中 A 是 $(n+m) \times n$ 的矩阵。

第二十步：如果把固定点设置为头 m 个顶点，那么线性系统如下。

$$\left[\begin{array}{c|c} L_u & 0 \\ \hline I_{m \times m} & 0 \end{array} \right] V'_d = \left[\begin{array}{c} 0 \\ V_{(1 \dots m)d} \end{array} \right]$$

第二十一部：可以用如下方式求解。

$$V'_d = (A^T A)^{-1} A^T b$$

第二十二步：从上述表示可以看出，线性系统的前 n 行是拉普拉斯坐标限制条件，后 n 行是欧几里得坐标限制条件。

第二十三步：上述系统求得的结果是光滑的，不能保持细节，除了已知的控制顶点周围的地方，同时把每个顶点向邻居顶点的中心进行移动。

第二十四步：对于平均拉普拉斯坐标来说，有一个切线方向的分量，而余切拉普拉斯坐标没有这个分量。

第二十五步：因此优化的核心思想是把这个分量移除，同时保留余切拉普拉斯法向方向上原有的细节。

第二十六步：方程的左边使用平均拉普拉斯矩阵，而方程的右边采用的是余切拉普拉斯坐标，从而得到下面的方程组。

$$\left[\begin{array}{c|c} L_u & \Delta_{d,c} \\ \hline I_{m \times m} & 0 \end{array} \right] V'_d = \left[\begin{array}{c} \Delta_{d,c} \\ V_{(1 \dots m)d} \end{array} \right]$$

第二十七步：如果把三维模型所有顶点的欧几里得坐标和位置坐标都作为限制条件，那么整个系统变为一个 $2n \times n$ 的系统。

$$A V'_d = b$$

第二十八步：加上每种限制条件的权重，扩展开来如下。

$$\left[\begin{array}{c} W_L L \\ W_P \end{array} \right] V'_d = \left[\begin{array}{c} W_L f \\ W_P V_d \end{array} \right]$$

第二十九步：这个系统可以用来改变顶点的位置，改变的结果既可以是优化原来的三维模型，也可以是使原来的模型光滑。

第三十步：假如把 $L = L_u$ 和 $f = \Delta_{d,c}$ ，那么求得的结果是使模型在保持原来的细节上进行优化。

第三十一部：假如 $L = L_{ck}$ 或 L_c ，或者 $L = L_u$ ，同时 $f = 0$ ，那么求得的结果使原来的模型光滑。

7.2.3 优化算法核心代码

上述系统中，需要设置顶点欧几里得坐标和拉普拉斯坐标两大类限制条件的权重，最简单的权重为：

$$W_L = I, W_P = W_{const} = sI$$

这个系统的算法和变形的算法类似，采用同样的框架，不同的是构建系统的部分，下面是不同部分的代码。

(1) 构建系统左边上部分矩阵：

```

public override SparseMatrix BuildMatrixA()
{
    SparseMatrix A = LaplaceManager. Instance. BuildMatrixCombinatorialGraphNormalized( mesh );
    return A;
}

```

(2) 用余切拉普拉斯坐标构建系统右边上部分向量，也就是所有顶点的拉普拉斯坐标限制条件：

```

public override double[ ][ ] BuildLaplacian()
{
    double[ ][ ] lap = LaplaceManager. Instance. ComputeLaplacianCotNormalize( mesh );
    return lap;
}

public override double[ ][ ] BuildBUPart()
{
    double[ ][ ] lap = BuildLaplacian();
    double[ ][ ] b = new double[3][ ];
    int n = MatrixA. RowSize;
    for( int i = 0; i < 3; i ++ )
    {
        b[i] = new double[ n ];
        for( int j = 0; j < mesh. Vertices. Count; j ++ )
        {
            b[i][j] = lap[i][j];
        }
    }
    lap = null;
    System. GC. Collect();
    return b;
}

```

(3) 构建系统右边下半部的向量，也就是所有顶点的位置限制条件：

```

public override void UpdateHandle()
{
    int k = mesh. Vertices. Count;
    for( int i = 0; i < mesh. Vertices. Count; i ++ )
    {
        b[0][k] = mesh. Vertices[i]. Traits. Position. x
                * ConfigLaplace. Instance. PositionWeight;
        b[1][k] = mesh. Vertices[i]. Traits. Position. y
                * ConfigLaplace. Instance. PositionWeight;
        b[2][k] = mesh. Vertices[i]. Traits. Position. z
    }
}

```

```
        * ConfigLaplace. Instance. PositionWeight;
        k++;
    }
}
```

(4) 构建系统左边下半部分：

```
public override void BuildConstraints( SparseMatrix A)
{
    for( int i =0; i < mesh. Vertices. Count; i++)
    {
        A. AddRow();
        A. AddElement( A. RowSize - 1, i, ConfigLaplace. Instance. PositionWeight);
    }
    A. SortElement();
}
```

7.2.4 优化算法效果

三维模型优化算法可以作用在各种各样的三维模型上。通过调节不同的权重，可以在保持细节和进行优化两个目标之间进行平衡，如图 7-9 所示，随着权重的增加，更能够保持原来三维模型的形状。优化算法的效果和顶点的平均曲率有关系，在平均曲率比较大的地方，优化的效果就不太好。可以把曲率作为权重加入上述的系统，从而改善优化的效果。随着权重的不同，最终的效果也不同，对于某些模型，需要保持边缘的特征，就需要把位置的权重加大，也就是尽可能改变边界顶点的位置。如图 7-10 所示，圆柱三维模型的边界是一个特征线，假如这个边界特征线的权重小，优化后边界就消失了，变得比较光滑。

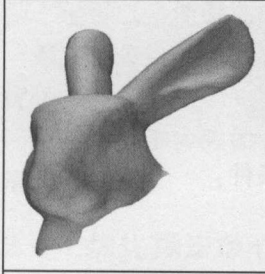
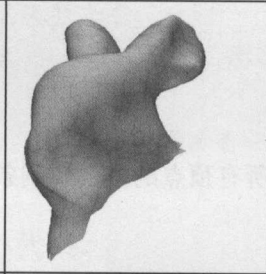
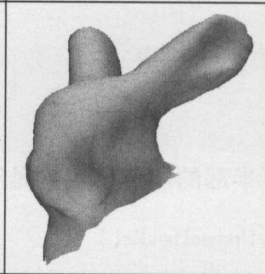
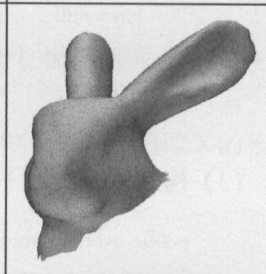
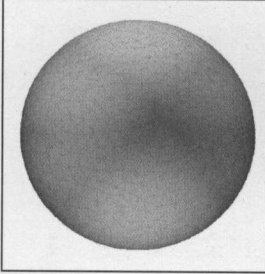
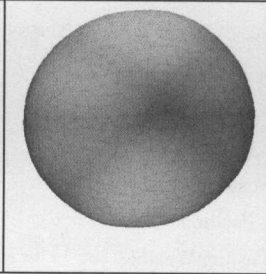
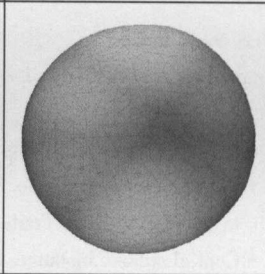
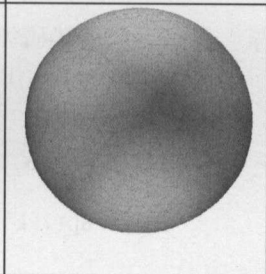
原始模型	权重0.01	权重0.1	权重1.0
			
			

图 7-9 不同权重的拉普拉斯模型优化效果

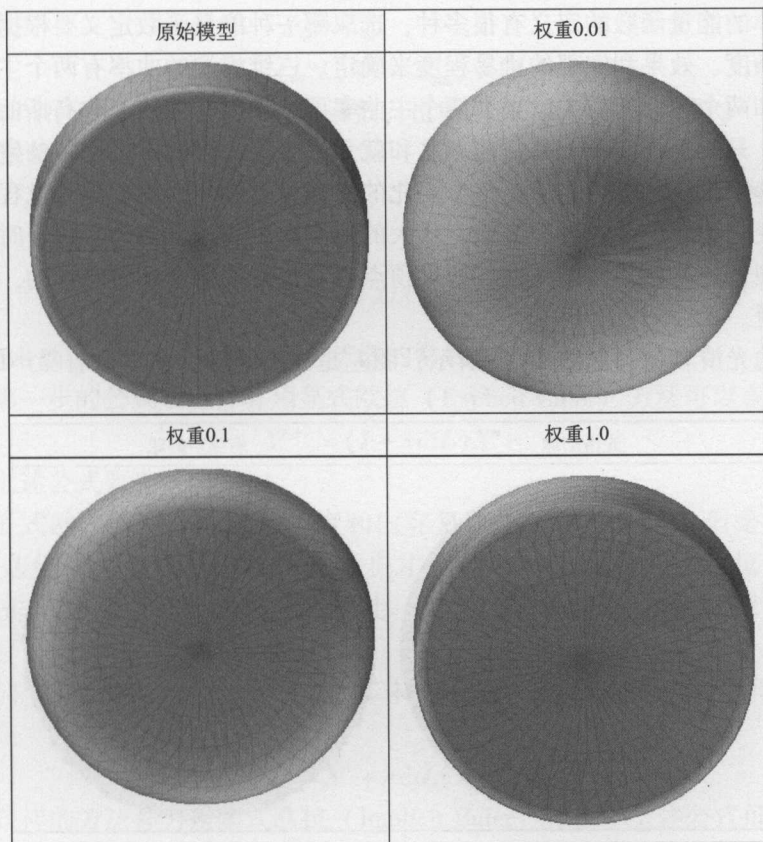


图 7-10 圆柱三维模型的拉普拉斯模型优化效果



7.3 拉普拉斯光滑算法

7.3.1 光滑算法介绍

三维模型的制作分为两大类，一类是通过专业的三维软件进行建模，如 3dsmax、maya、blender 等。另一类是通过三维扫描仪扫描得到。三维扫描仪可以把真实存在的物体扫描为虚拟的三维物体。用软件建模可以很方便地控制想要得到的三维模型。而用三维扫描仪扫描得到的三维模型在生成的时候有时候会有很多噪声。噪声指的是三维模型的表面会凹凸不平，这样的三维模型需要进行处理才能使用。模型的光滑指的就是通过算法把这些噪声去掉。光滑算法就是研究如何去除三维模型噪声的算法。有时候在三维模型处理的时候不仅仅希望去除噪声，也希望能够去掉某些三维模型上的细节，从而得到一个更加平滑的三维模型。但是在去掉噪声和细节的同时，还要求能够保持三维模型的形状和特征。

三维模型的光滑和粗糙体现在三维模型的曲率上。光滑算法的核心思路是减少三维模型的曲率，从而达到去除噪音的目的。因此需要定义一个和曲率有关的能量函数，然后改变这

个三维模型顶点的位置，使能量函数变小，从而可以减少曲率，也就是使三维模型变得比较光滑。基于曲率的能量函数的定义有很多种，选取哪一种能量函数定义要根据所得到的光滑算法的时间复杂度、效果和实现的难易程度来确定。三维模型的曲率有两个主曲率，因此能量函数的定义和两个主曲率有关。根据两个主曲率可以生成平均曲率和高斯曲率，但是无论模型如何光滑，只要拓扑不变，高斯曲率之和就无法改变，因此能量函数的定义不能和高斯曲率有关。在确定了能量函数之后，整个变化的过程可以看作为热传导的过程，也就是三维模型上的曲率变化和热传导的过程类似，从大的地方向小的地方传递，随着时间的推移，慢慢整个三维模型上的曲率趋向于一致。如果顶点沿着拉普拉斯坐标方向移动，这种方法也称为拉普拉斯光滑。

三维模型的光滑和噪音如图 7-11 所示，图左是光滑的球，图右是有噪声的球。

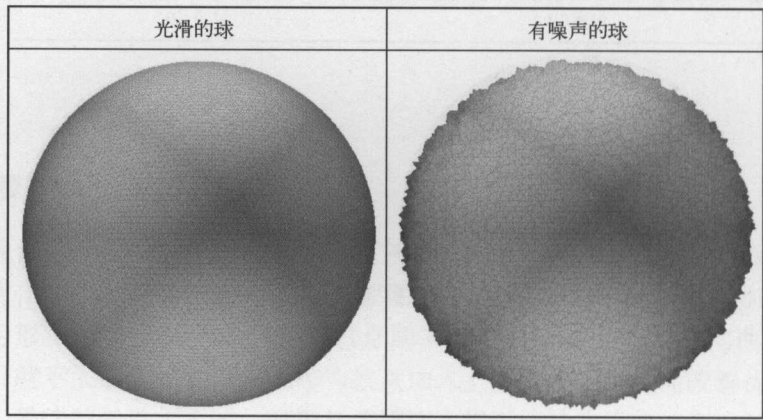


图 7-11 有噪声的球形模型

7.3.2 能量函数和数学系统

三维模型光滑的思路是定义一个能量函数来指导光滑的过程，这个能量函数表示了三维模型的噪音程度。通过这个能量函数可以度量三维模型的造影程度。在有了能量函数之后，再对能量函数求导，从而的能量函数的最小值。三维模型光滑算法的数学模型就建立在这个能量函数之上。最终还是得到和拉普拉斯矩阵相关的一个数学模型。在具体的数值计算中，可以采用显式欧拉或者隐式欧拉的方法得到稳定的数值解。

1. 能量函数

第一步：定义光滑的能量函数，常用的能量函数的定义有三种，如下所示。

$$E(S) = \int_S \kappa_1^2 + \kappa_2^2 dS$$
$$E_{membrane}(X) = \frac{1}{2} \int_{\Omega} X_u^2 + X_v^2 dudv$$
$$E_{thin\ plate}(X) = \frac{1}{2} \int_{\Omega} X_{uu}^2 + 2X_{uv}^2 + X_{vv}^2 dudv$$

第二步： κ_1 和 κ_2 是两个主曲率，第一种能量函数定义了总曲率。但是这个函数和顶点的坐标是非线性关系，因此在算法计算中一般不采用。

第三步：第二个和第三个能量函数求导后分别如下。

$$L(X) = X_{uu} + X_{vv}$$

$$L^2(X) = L^{\circ}L(X) = X_{uuuu} + 2X_{uuvv} + X_{vvvv}$$

第四步：本节中采用的是第二种能量函数，也就是采用一阶拉普拉斯。

2. 数学系统构建

第一步：让 X 表示三维模型的顶点坐标，顶点变化的过程是一个热扩散的过程，那么扩散的公式如下所示。

$$\frac{\partial X}{\partial t} = \lambda L(X)$$

第二步：上一步变化公式中用 λ 表示扩散的常数参数， t 是时间。

第三步：第一步的公式数值计算用显式欧拉（Explicit Euler）方法可以表示为，

$$X^{n+1} = (I + \lambda dt L) X^n$$

第四步：上述公式要满足 $\lambda dt < 1$

第五步：显式欧拉方法是时间复杂度和内存复杂度都是线性的，但是当模型比较大的时候，由于迭代条件的限制，需要数百步才能迭代收敛。在显式欧拉方法中，能量函数的导数，也就是拉普拉斯矩阵是通过上一步的模型求得的，因此稳定性和有效性比较差。

第六步：如果拉普拉斯矩阵是通过下一步未知的模型来求，那么就可以得到无条件稳定的系统。也就是

$$X^{n+1} = X^n + \lambda dt L(X^{n+1})$$

第七步：这样的方法称为是隐式欧拉（Implicit Euler）方法。上述公式可以表示为

$$(I - \lambda dt L) X^{n+1} = X^n$$

第八步：这样每一步就不需要受 $\lambda dt < 1$ 的限制条件限制。因此可以用很大的参数进行求解，但是相对于显式欧拉方法，隐式欧拉需要解一个线性方程组。

第九步：拉普拉斯矩阵 $A = I - \lambda dt L$ 是稀疏的，平均每行只有 6 个非零值，因此线性方程可以很快求解。

第十步：在上述算法中，所用到的拉普拉斯矩阵可以表示为如下形式。

$$\sum_{v_j \in N(v_i)} \frac{1}{w_{ij} v_j} \sum_{w_{ij} v_j \in N(v_i)} w_{ij} (p_j - p_i)$$

第十一步：在这个拉普拉斯矩阵中，值得注意的是对于显式欧拉防范没有把面积计算进去。但是在隐式欧拉方法中，可以把面积计算进去。

第十二步：如果采用平均拉普拉斯矩阵，让 m 表示一个顶点的邻居顶点数量，那么

$$w_{ij} = 1$$

$$L(x_i) = \frac{1}{m_{j \in N_1(i)}} \sum x_j - x_i$$

第十三步：也可以采用余切拉普拉斯权重，也就是

$$w_{ij} = \cot \alpha_{ij} + \cot \beta_{ij}$$

α_{ij} 和 β_{ij} 如图 7-12 所示。

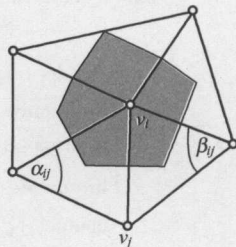


图 7-12

第十四步：对于隐式欧拉方式求解，可以表示为

$$X^{n+1} = (I - \lambda dt L)^{-1} X^n$$

第十五步：用泰勒展开，变为

$$I + \lambda dt L + (\lambda dt)^2 L^2 + \dots$$

第十六步：从中可以看出隐式欧拉方法相当于显式欧拉方法加上无数个高阶的拉普拉斯光滑。

第十七步：这个光滑算法，其实就是把每个顶点沿着法向方向朝着三维模型内部进行移动，因此会造成三维模型的缩小。因此需要重新缩放三维模型的顶点位置，从而保持整个体积不变。

第十八步：用 V^0 和 V^n 分别表示三维模型起始体积和第 n 步的体积，对于第 n 步得到的三维模型每个顶点乘上如下缩放因子，就可以保持体积不变。

$$\beta = (V^0/V^n)^{1/3}$$

第十九步：上述的曲率扩散过程和曲率流类似，曲率流定义如下。

$$\frac{\partial x_i}{\partial t} = -\bar{\kappa}_i n_i$$

第二十步：其中的曲率是平均曲率。

$$\bar{\kappa} = (\kappa_1 + \kappa_2)/2$$

第二十一部：平均曲率可以用如下公式计算。

$$-\bar{\kappa}n = \frac{1}{4A} \sum_{i \in N(i)} (\cot \alpha_j + \cot \beta_j) (x_j - x_i)$$

第二十二步：从而假如拉普拉斯矩阵采用平均曲率矩阵，那么隐式欧拉方法扩散过程就是平均曲率流过程。

7.3.3 光滑算法核心代码

光滑算法的核心代码分为显式欧拉方法和隐式欧拉方法两种。显式欧拉方法计算速度快，但是不稳定，迭代比较慢。隐式欧拉方法需要求解线性方程组，相比较来说计算速度慢，但是比较稳定。迭代的次数比较少。

1. 显式欧拉方法

```
public static void SmoothTaubin(TriMesh Mesh)
{
    double weight = ConfigMeshOP.Instance.SmoothTaubinLamda;
    int iterative = ConfigMeshOP.Instance.SmoothTaubinIterative;
    bool cot = ConfigMeshOP.Instance.SmoothTaubinCot;
    int n = Mesh.Vertices.Count;
    double[,] lap = null;
    for(int j = 0; j < iterative; j++)
    {
```

```

        if( cot)
        {
            lap = LaplaceManager. Instance. ComputeLaplacianCotNormalize( Mesh );
        }
        else
        {
            lap = LaplaceManager. Instance. ComputeLaplacianUniform( Mesh );
        }
        for( int i = 0; i < n; i ++ )
        {
            Mesh. Vertices[ i ]. Traits. Position. x += lap[ 0 ][ i ] * weight;
            Mesh. Vertices[ i ]. Traits. Position. y += lap[ 1 ][ i ] * weight;
            Mesh. Vertices[ i ]. Traits. Position. z += lap[ 2 ][ i ] * weight;
        }
    }
}

```

2. 隐式欧拉方法

(1) 计算线性方程的矩阵，也就是 $I - \lambda dt L$

```

public override SparseMatrix BuildMatrixA()
{
    SparseMatrix A = null;
    if( ConfigLaplace. Instance. SmoothType == 1 )
    {
        A = LaplaceManager. Instance. BuildLaplaceTutte( mesh );
    }
    else if( ConfigLaplace. Instance. SmoothType == 2 )
    {
        A = LaplaceManager. Instance. BuildMatrixCotNormalize( mesh );
    }
    else if( ConfigLaplace. Instance. SmoothType == 3 )
    {
        A = LaplaceManager. Instance. BuildMatrixMeanCurvature( mesh );
    }
    A. Scale( ConfigLaplace. Instance. SmoothWeight );
    SparseMatrix identity = SparseMatrix. Identity( A. RowSize );
    A = A. Add( identity );
    return A;
}

```

(2) 保持体积不变。

```

public void PreserveVolume()
{
    double oldVolume = TriMeshUtil. ComputeVolume( BackUpMesh );
    double newVolume = TriMeshUtil. ComputeVolume( mesh );
    for( int i = 0; i < mesh. Vertices. Count; i ++ )
    {
        mesh. Vertices[ i ]. Traits. Position. x * = Math. Pow( oldVolume/newVolume,
                                                                1/3 );
        mesh. Vertices[ i ]. Traits. Position. y * = Math. Pow( oldVolume/newVolume,
                                                                1/3 );
        mesh. Vertices[ i ]. Traits. Position. z * = Math. Pow( oldVolume/newVolume,
                                                                1/3 );
    }
}

```

(3) 构建方程组的右边向量。

```

public override double[ ][ ] BuildBUpPart()
{
    double[ ][ ] b = new double[3][ ];
    int n = MatrixA. RowSize;
    for( int i = 0; i < 3; i ++ )
    {
        b[ i ] = new double[ n ];
    }
    for( int j = 0; j < mesh. Vertices. Count; j ++ )
    {
        b[ 0 ][ j ] = mesh. Vertices[ j ]. Traits. Position. x;
        b[ 1 ][ j ] = mesh. Vertices[ j ]. Traits. Position. y;
        b[ 2 ][ j ] = mesh. Vertices[ j ]. Traits. Position. z;
    }
    return b;
}

```

7.3.4 光滑算法效果展示

光滑算法可以作用在各种不同形状的三维模型上，都可以得到很好的效果。但是光滑算法在删除噪音的同时，有可能会把细节也删除，就造成原来的三维模型变得更细。

(1) 这个实验中，采用的是平均拉普拉斯矩阵，算法效果如图 7-13 所示。

(2) 这个实验中采用的是余切权重拉普拉斯矩阵，算法效果如图 7-14 所示。

(3) 假如一个模型的采样步均匀，那么用面积余切拉普拉斯隐式方法可以保持原来形状不变，如图 7-15 所示。

(4) 通过平均曲率的显示可以看出，光滑后的曲率比原来变小，如图 7-16 所示。

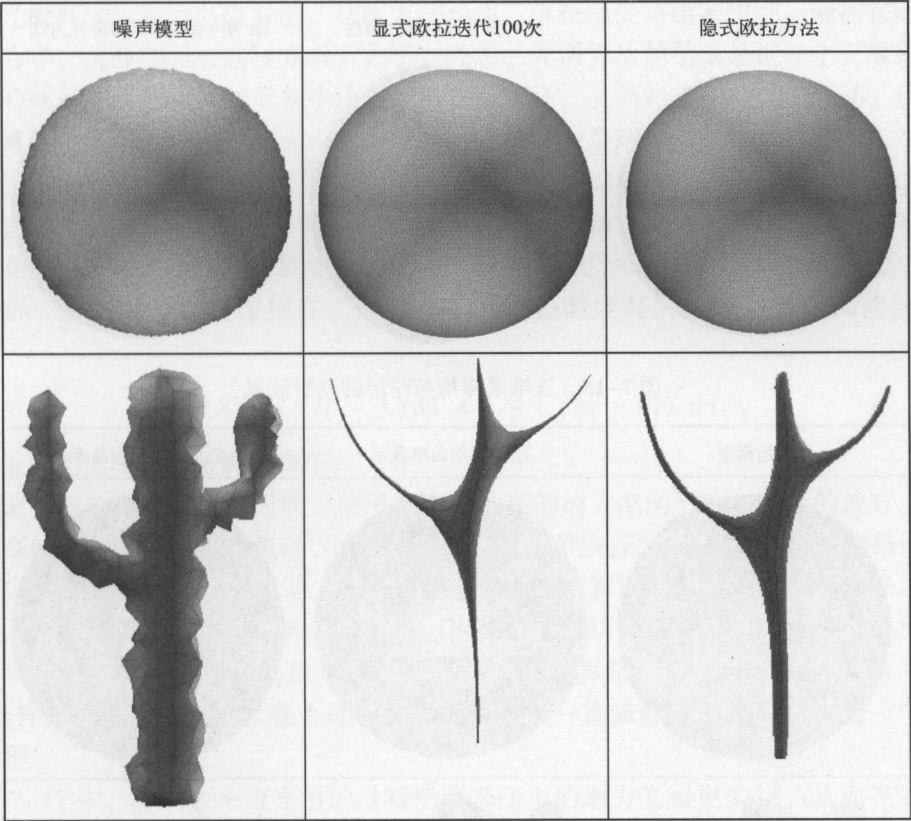


图 7-13 拉普拉斯光滑算法效果

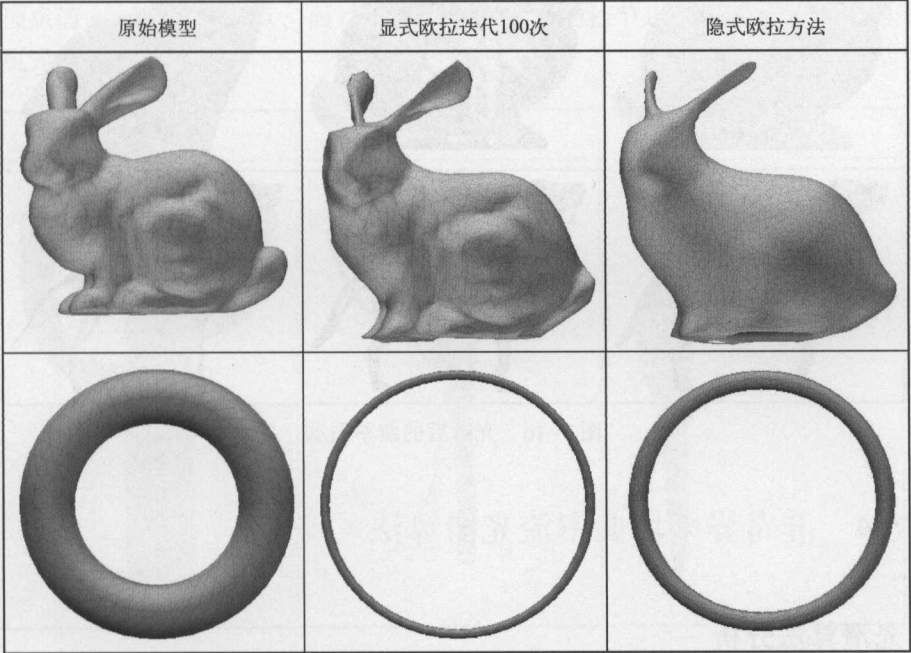


图 7-14 两种欧拉方法的光滑效果

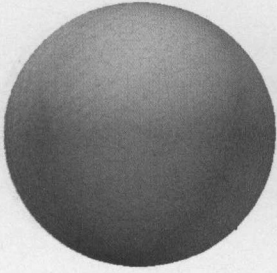
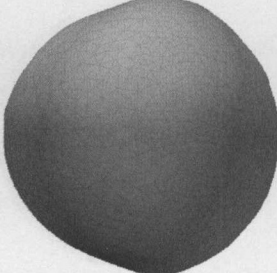
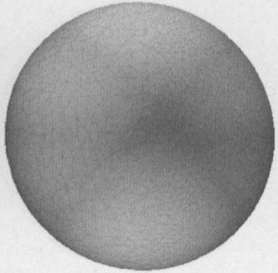
原始模型	平均拉普拉斯显式方法	余切拉普拉斯矩阵隐式方法
		

图 7-15 三维采样均与球形的光滑效果

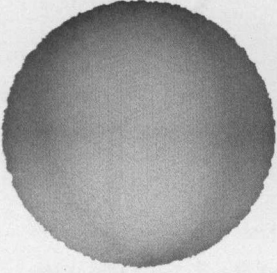
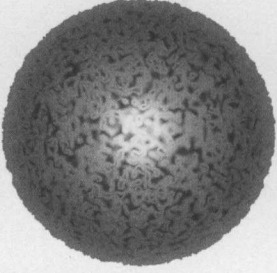
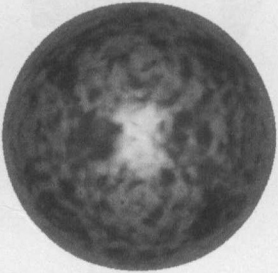
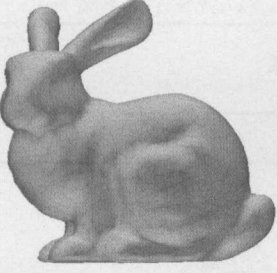
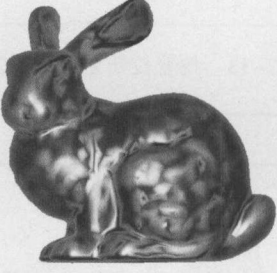
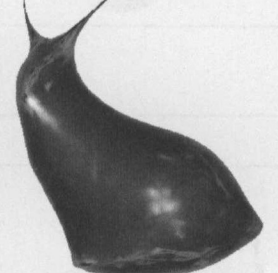



原始模型	原始平均曲率显示	光滑后的平均曲率
		
		
		

图 7-16 光滑后的曲率图示



7.4 非奇异平均曲率流光滑算法

7.4.1 光滑算法分析

三维模型光滑的过程是一个类似于流的过程，顶点的曲率从高的地方流到低的地方。从

上一节的光滑算法效果中可以看出，这些方法会使三维模型变得越来越细，最后在某个部位的体积变为零，这样就无法继续光滑下去了。理想的光滑算法的结果是把一个三维模型变为一个光滑的球形。因为一个球形各个顶点的曲率都一样，从而曲率不再发生流动。但是上一节的光滑算法无法把三维模型变为一个球形或者近似于一个球形。

采用平均曲率使模型光滑的方法称为平均曲率流（Mean Curvature Flow）MCF 方法。除了平均曲率流 MCF，还有里奇流（Ricci Flow）、威廉姆流（William Flow）等方法使模型光滑。里奇流方法作用在三维模型的内蕴属性上，得到的是三维模型光滑后的度量，而不是坐标。威廉姆流也是作用在三维模型上，但是需要高阶导数。威廉姆流能量函数定义为

$$E_w(S) = \int_S (H^2 - K) dA = 1/4 \int_S (\kappa_1 - \kappa_2)^2 dA$$

其中， H 是平均曲率， K 是高斯曲率。

平均曲率流方法在光滑的时候，使平均曲率从高的地方流向平均曲率低的地方。随着平均曲率从高的顶点向低的顶点流动，会在三维模型凸出的地方形成奇异值，也就是使凸起的地方特别尖锐，曲率无法继续流动，从而阻止平均曲率流算法的继续进行。因此平均曲率流的算法不能使三维模型变为一个光滑的球形。因此需要分析产生奇异值的条件，从而修改算法，使平均曲率流方法可以变为保角平均曲率流 CMCF 算法（Conformalized Mean Curvature Flow）。这样的修改使曲率可以避免奇异值，从而可以一直流动下去，直至成为一个球形或者近似球形。

在图 7-17 中，平均曲率流光滑的过程把原来凸出的地方变得更尖锐，从而在系统中产生奇异值。例如，仙人掌的枝叶部分变为一根体积几乎为零的细线，从而在数值计算上无法继续使曲率进行流动。而保角平均曲率流的光滑算法在光滑的过程中，不仅仅在局部上使曲率从高的顶点流向周围低的顶点，而且在整体上仙人掌的枝节也收缩回去，慢慢消失，从而得到一个更光滑的结果。

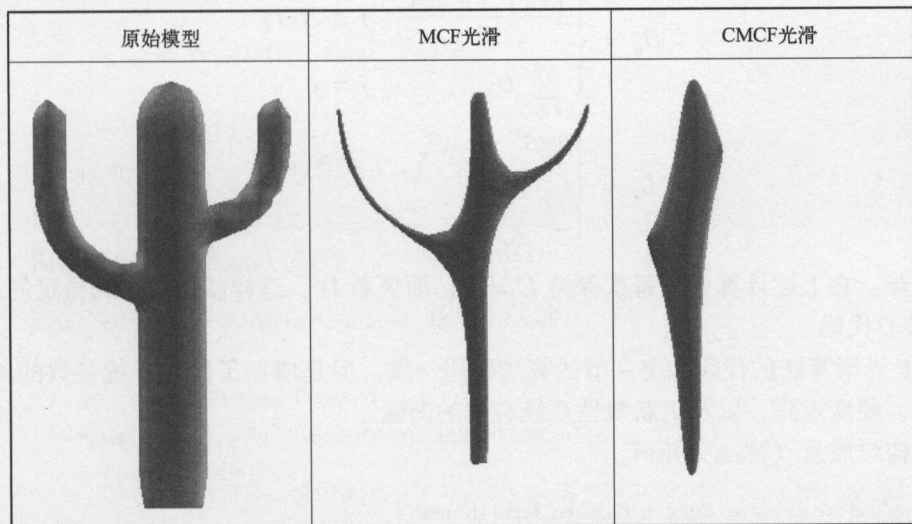


图 7-17 光滑效果对比

7.4.2 数学推导和核心代码

1. 推导过程

CMCF 报角平均曲率流光滑算法的构建过程主要是基于数值分析，也就是在原来的平均曲率流的数学系统上，分析产生奇异值的原因，然后针对这些原因，更改系统的参数，从而使系统可以具有稳定的解，避免奇异值的出现。

第一步：如果三维模型表示为

$$\Phi_t: M \rightarrow \mathbb{R}^3$$

那么平均曲率流表示为

$$\frac{\partial \Phi_t}{\partial t} = \Delta_t \Phi_t$$

第二步：上述方程是光滑曲面上的方程，离散化需要一个函数基

$$\{B_1, \dots, B_N\}: M \rightarrow \mathbb{R}$$

那么这个方程可以离散化为

$$\int_M \left(\frac{\partial \Phi_t}{\partial t} \cdot B_i \right) d\mu_t = \int_M (\Delta_t \Phi_t \cdot B_i) d\mu_t \quad \forall 1 \leq i \leq N$$

第三步：用 D^t 和 L^t 分别表示为质量 (Mass) 和硬度 (Stiffness) 矩阵，分别为

$$D_{ij}^t = \int_M (B_i \cdot B_j) d\mu_t$$

$$L_{ij}^t = - \int_M g_t (\nabla_t B_i, \nabla_t B_j) d\mu_t$$

那么平均曲率流两步之间的关系表达为

$$(D^t - \delta L^t) \vec{x}(t + \delta) = D^t \vec{x}(t)$$

第四步：矩阵形式为：

$$D_{ij} = \begin{cases} \frac{|T_{ij}^1| + |T_{ij}^2|}{12} & j \in N(i) \\ \sum_{k \in N(i)} D_{ik} & j = i \end{cases}$$

$$L_{ij} = \begin{cases} \frac{\cot \beta_{ij}^1 + \cot \beta_{ij}^2}{2} & j \in N(i) \\ - \sum_{k \in N(i)} L_{ik} & j = i \end{cases}$$

第五步：在上述计算中，每次保持 $L^t = L^c$ ，而更新 D^t ，这样就能够得到稳定的系统。

2. 核心代码

CMCF 光滑算法的代码和上一节的算法代码一致。但是增加了修改系统参数的函数，如质量矩阵、硬度矩阵，以及更新线性系统右边的向量。

(1) 构建质量 (Mass) 矩阵。

```
public SparseMatrix BuildMatrixMass(TriMesh mesh)
{
    int n = mesh.Vertices.Count;
```

```

SparseMatrix L = new SparseMatrix(n,n);
for( int i = 0; i < mesh. Edges. Count; i ++ )
{
    double face0area = 0;
    double face1area = 0;
    if( mesh. Edges[ i ]. Face0 ! = null )
    {
        face0area = TriMeshUtil. ComputeAreaFace( mesh. Edges[ i ]. Face0 );
    }
    if( mesh. Edges[ i ]. Face1 ! = null )
    {
        face1area = TriMeshUtil. ComputeAreaFace( mesh. Edges[ i ]. Face1 );
    }
    L. AddValueTo ( mesh. Edges[ i ]. Vertex0. Index,
                    mesh. Edges[ i ]. Vertex1. Index,
                    ( face0area + face1area ) / 12 );
    L. AddValueTo ( mesh. Edges[ i ]. Vertex1. Index,
                    mesh. Edges[ i ]. Vertex0. Index,
                    ( face0area + face1area ) / 12 );
}
for( int i = 0; i < n; i ++ )
{
    double sum = 0;
    foreach( SparseMatrix. Element e in L. Rows[ i ] )
    {
        sum += e. value;
    }
    L. AddValueTo( i, i, sum );
}
L. SortElement();
return L;
}

```

(2) 构建硬度 (Stiffness) 矩阵。

```

public SparseMatrix BuildMatrixStiffness( TriMesh mesh )
{
    int n = mesh. Vertices. Count;
    SparseMatrix L = BuildLaplaceMatrixCotBasic( mesh );
    for( int i = 0; i < n; i ++ )
    {
        double sum = 0;
        foreach( SparseMatrix. Element e in L. Rows[ i ] )

```

```

    {
        e.value = e.value/2;
        sum += e.value;
    }
    L.AddValueTo(i,i, -sum);
}
L.SortElement();
return L;
}

```

(3) 构建线性系统矩阵。

```

public override SparseMatrix BuildMatrixA()
{
    SparseMatrix A = LaplaceManager.Instance.BuildMatrixCot(BackUpMesh);
    SparseMatrix massD = LaplaceManager.Instance.BuildMatrixMass(mesh);
    A.Scale(ConfigLaplace.Instance.CMCFStep);
    A = A.Add(massD);
    return A;
}

```

(4) 构建线性系统右边。

```

public double[][] ComputeLaplacianCMCF(TriMesh mesh)
{
    SparseMatrix L = this.BuildMatrixMass(mesh);
    double[][] lap = ComputeLaplacianBasic(L, mesh);
    return lap;
}

```

7.4.3 CMCF 光滑算法效果展示

CMCF 算法可以作用于各种各样的三维模型，如兔子模型，有边界的、没有边界的模型，以及各种拓扑的模型。这些模型假如没有边界，光滑的结果最终会变为一个椭球，假如

有边界，会变为一个椭圆。

1. 实验一

图 7-18 是一个模型在 CMCF 流下光滑的过程，可以看出凸出的部分最后都会被光滑，如兔子的耳朵部分。

2. 实验二

CMCF 算法在有边界的模型上进行光滑，可以把这个三维模型逐渐变为一个椭圆，如图 7-19 所示，三维模型上的细节慢慢减少，在第十八次迭代的时候就变为一个椭圆。

3. 实验三

在图 7-20 的机械模型中，具有棱角分明的边缘。CMCG 算法可以把棱角逐渐变光滑，从而使机械模型的棱角消失，最终也变为一个椭球。

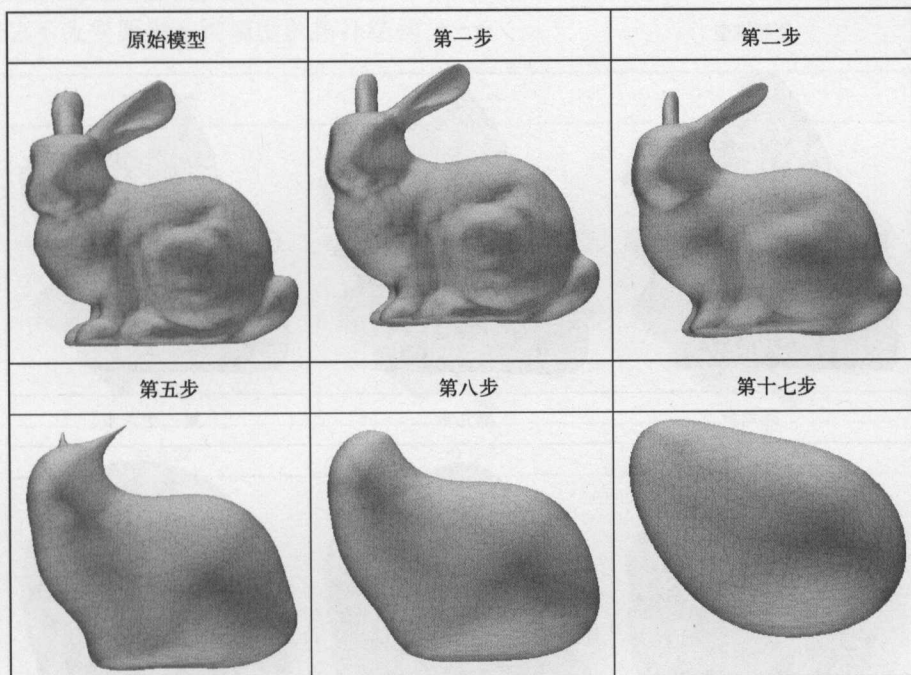


图 7-18 兔子模型的光滑过程

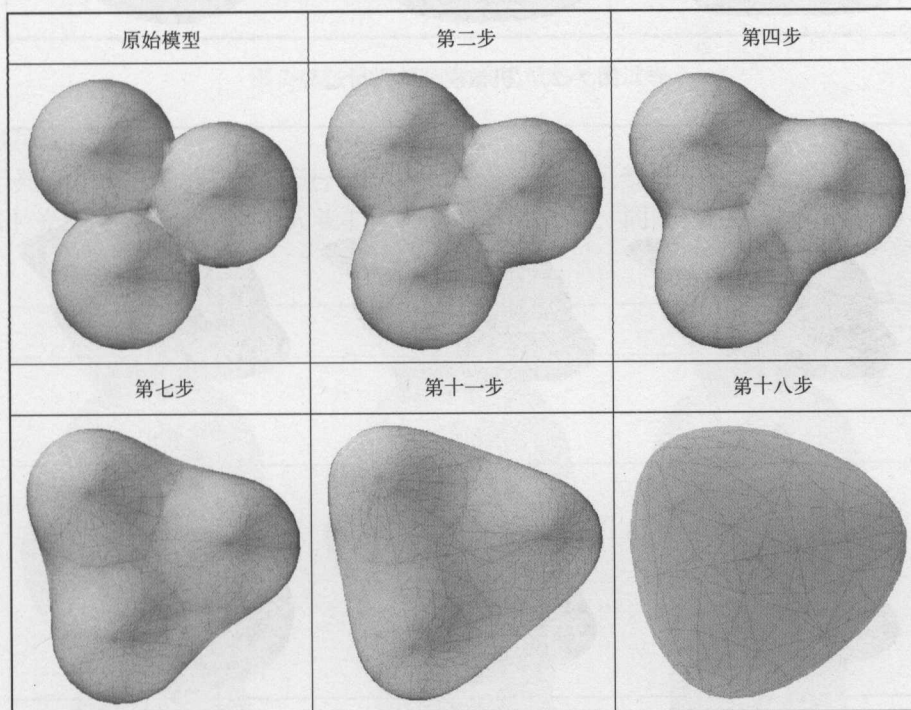


图 7-19 有边界的三维模型光滑过程

4. 实验四

图 7-21 中的三维模型具有比较复杂的表面形状, 采用 CMCF 算法也可以成功使它变为一个椭球。

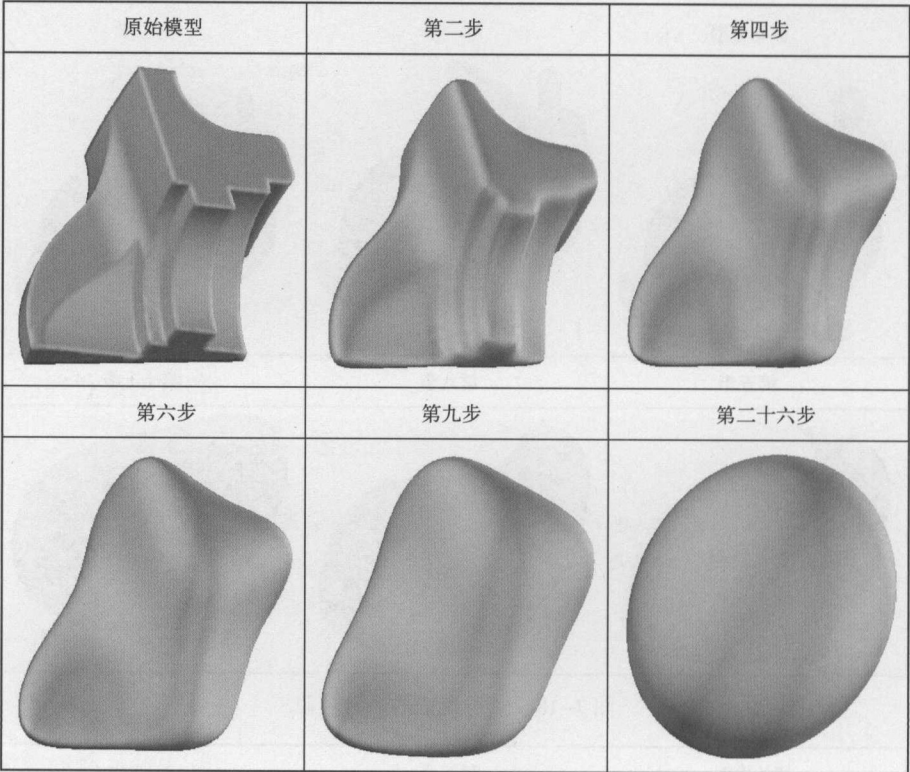


图 7-20 机械模型的光滑过程

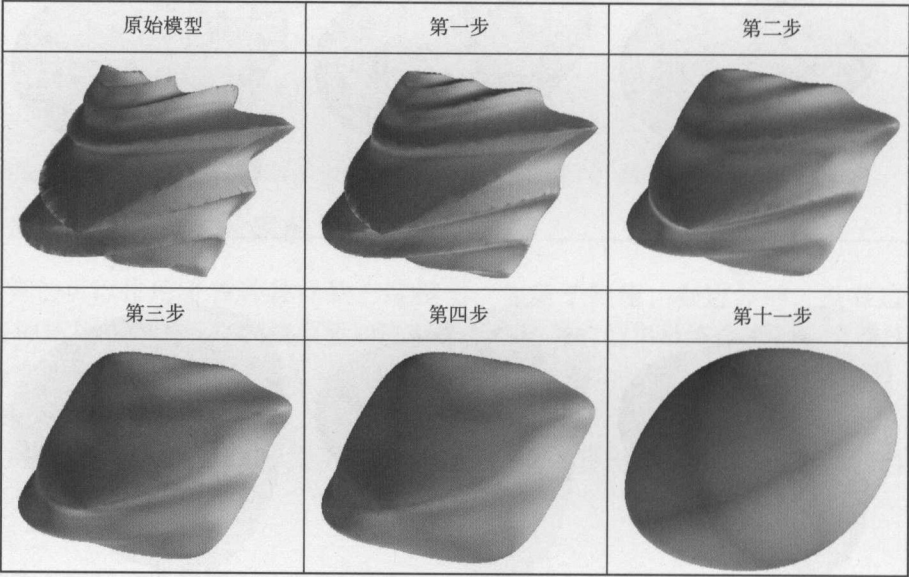


图 7-21 复杂模型的光滑过程

5. 实验五

在图 7-22 中，三维模型具有不同的拓扑结构，并且表明有锯齿。CMCF 算法经过十三次迭代可以把这个拓扑结构的三维模型变为一个光滑的拓扑相同的圆环。从中可以看出，

CMCF 算法不改变原来三维模型的拓扑结构。

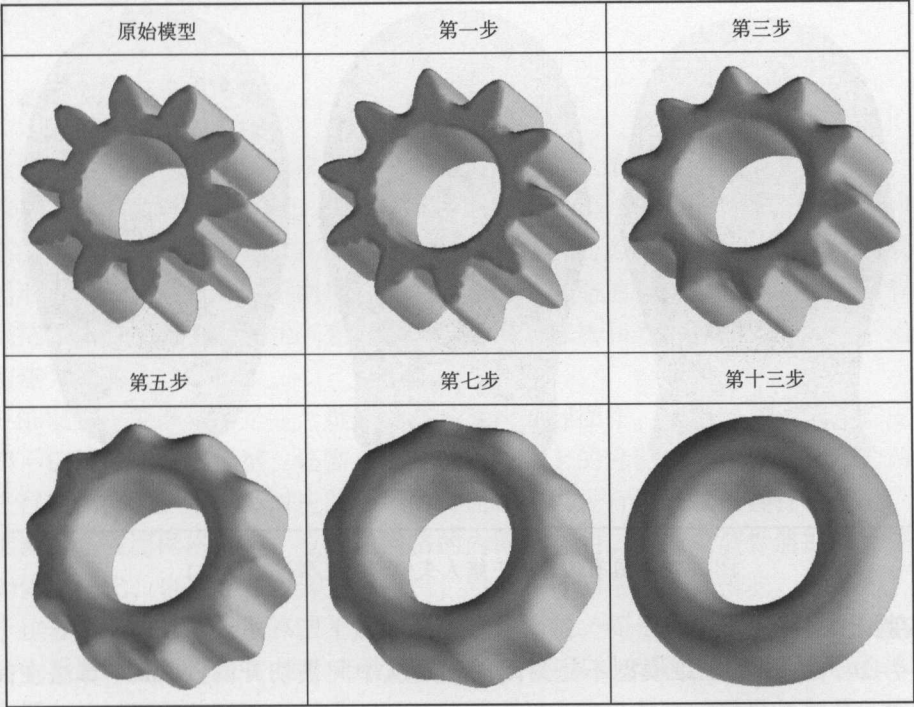


图 7-22 复杂拓扑结构模型的光滑过程

6. 实验六

对于具有多个边界的三维模型，如图 7-23 人头三维模型所示，具有嘴巴和脖子两个边界。CMCF 光滑算法可以把这个人头变为一个光滑的人头，同时嘴巴和脖子的边界部位变为一个圆。

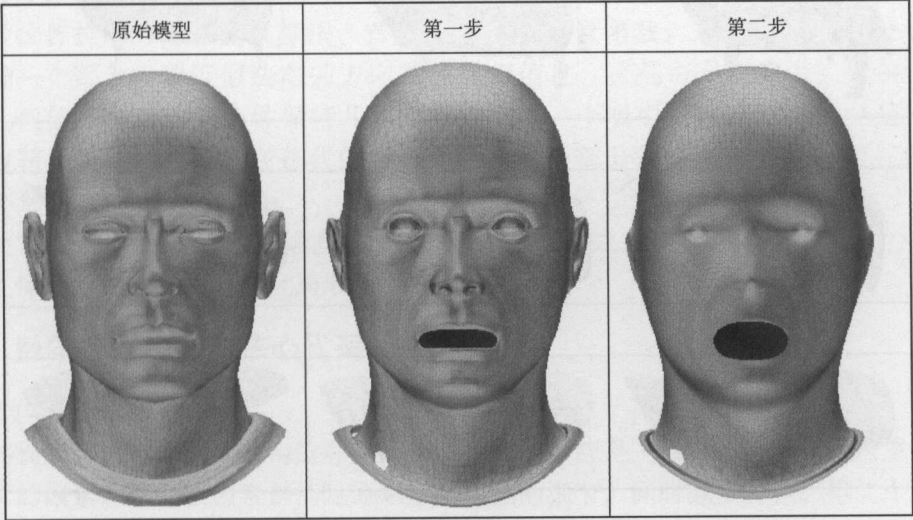


图 7-23 具有边界的三维人头模型的光滑过程

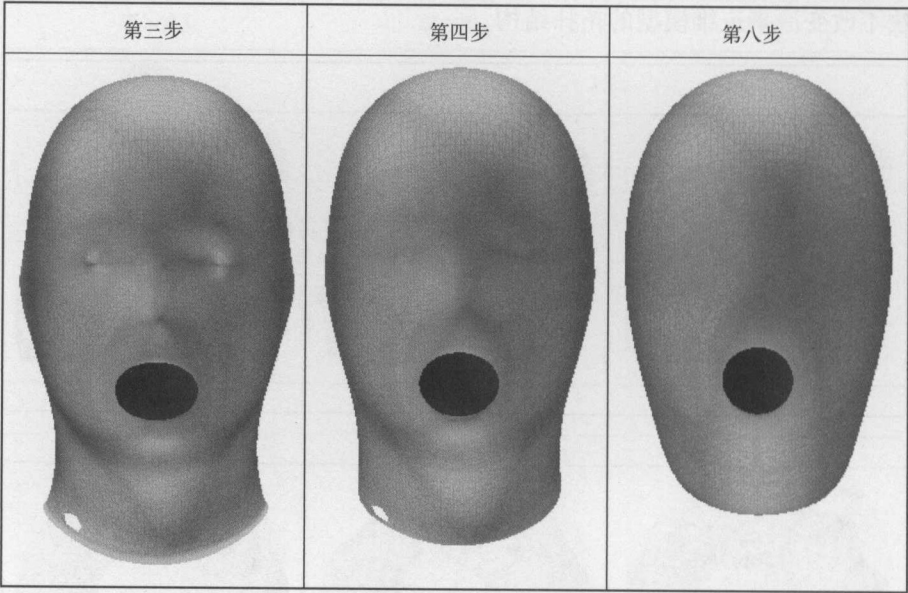


图 7-23 具有边界的三维人头模型的光滑过程（续）

7. 实验七

在图 7-24 中，牛的三维模型不是封闭的，是从中间被切开的。CMCF 算法在前三步光滑的过程中，牛头和牛脚的部位出现比较尖锐的部分。但是由于算法在设计的时候已经考虑了奇异值的情况，并且可以通过调整参数来避免奇异值，因此随后的几步中，尖锐的部位慢慢消失，最后得到一个光滑的圆盘模型。

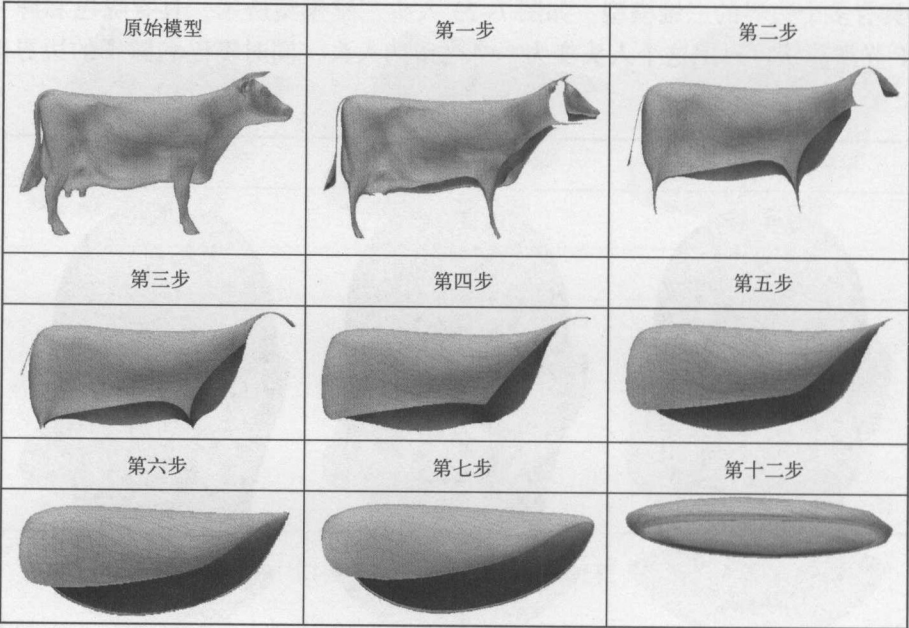


图 7-24 三维牛模型的光滑过程



7.5 骨骼抽取

7.5.1 骨骼抽取概述

三维模型光滑目标是使三维模型的细节都消失，使三维模型的表面变得光滑。而光滑相反的操作是三维模型的骨骼抽取。骨骼抽取是从三维模型中得到能够代表三维模型形状的、体积为零的骨骼。三维模型光滑算法把所有三维模型都变为一个球形，而骨骼抽取算法针对每个不同形状的三维模型得到不同形状的骨骼。虽然骨骼抽取算法也使局部的细节消失，但是需要保持全局的、整体的形状不变。虽然骨骼抽取算法和光滑算法的目标大不相同，但是都可以构建为类似的数学框架。

从三维模型中抽取骨骼是一个重要的三维模型处理操作。因为骨骼动画、蒙皮等三维动画技术都不仅仅需要三维模型，还需要这个三维模型上的骨骼，从而通过骨骼的动作驱动三维模型进行动作。和三维模型对应的一维的骨骼还可以进行三维模型查找、动画、变形等操作。三维模型是表面网格模型，也就是网格的内部是空的。三维模型骨骼提取算法通常是把三维模型体网格化，也就是在内部生成网格，然后在体网格上进行提取。

基于拉普拉斯的骨骼提取算法不需要先生成体网格，而是直接作用于表面网格模型。这个算法的主要思路是通过拉普拉斯光滑的框架在一些限制条件下使三维模型的体积缩小为零。这个提取的过程不改变原来模型的特点和模型的连接关系。然后再从体积为零的三维模型中得到骨骼，骨骼用一维的线条来表示。这个算法的核心要点是在体积缩小的时候，能够保持整体的形状不变，这和上一节 CMCF 的算法目标正好相反，上一节的目标是避免体积缩小为零，而骨骼提取的目标是尽量使体积为零。CMCF 流的过程是使形状变为球，原来的三维模型的特征都消失了。骨骼提取算法是基于光滑的框架，因此即使原来模型上有噪声，也可以成功提取骨骼。并且采用的是隐式欧拉方法，从而数值计算稳定性和可靠性，收敛性都很好。并且在提取的同时也建立了骨骼和顶点的对应关系。

基于拉普拉斯的骨骼提取算法，首先构造一个能量函数：这个能量函数分为两个部分，其中一个部分沿着近似法向的方向移除几何信息，这是一个收缩力；另一个部分用顶点的几何信息作为限制条件保持几何信息，这是一个吸引力。这两个部分是互相矛盾的，通过精心设计的权重，在迭代的过程中，使两个部分保持平衡，从而使三维模型收缩为骨骼形状。

如图 7-25 所示，仙人掌三维模型经过骨骼提取算法之后变为体积几乎为零的线条，但是线条还保持原来三维模型的全局形状。

7.5.2 数学模型构建和核心代码

1. 数学推导

骨骼提取算法数学模型的构建和光滑算法的数学系统是一致的，但是需要考虑的是两个互相平衡的因素。第一个因素是消除原来三维模型的细节，同时减少体积，第二个因素是保持原来三维模型的形状不变。这两个因素的目标是相反的，因此需要互相平衡。因此需要设置一些参数作为权重使两个因素能够并行实现。

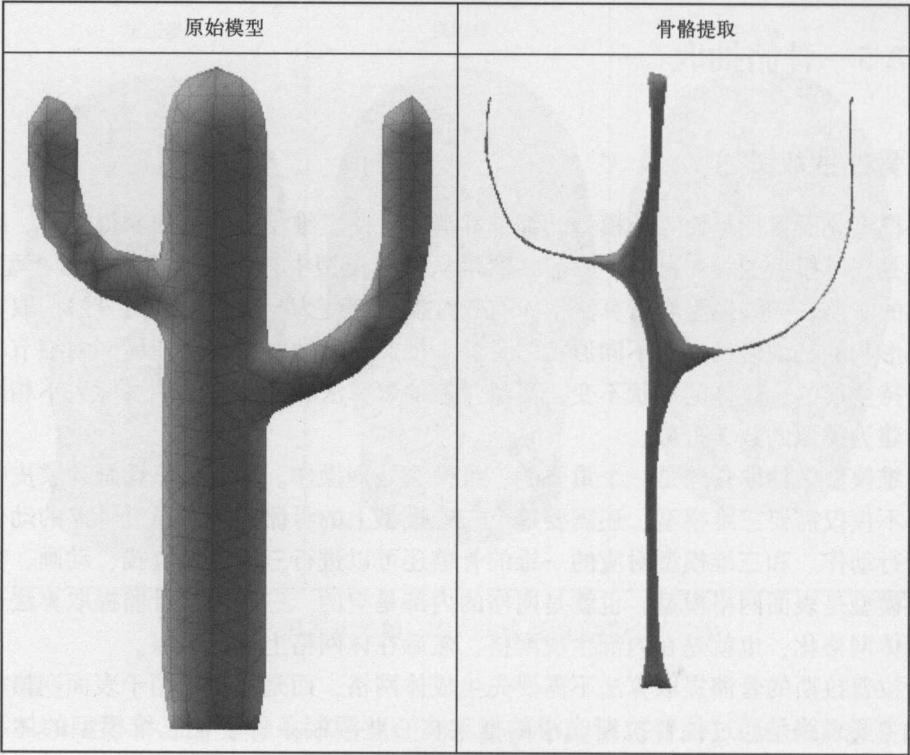


图 7-25 仙人掌骨骼抽取

第一步：让 V' 表示光滑后的顶点，那么构造如下线性系统。

$$LV' = 0$$

第二步：其中拉普拉斯矩阵采用余切权重。

$$L_{ij} = \begin{cases} w_{ij} = \cot\alpha_{ij} + \cot\beta_{ij} & (i,j) \in E \\ \sum_{(i,k) \in E}^k -w_{ik} & i = j \\ 0 & \text{其他} \end{cases}$$

第三步：余切拉普拉斯坐标近似于平均曲率和法向的乘积，也就是

$$\delta = LV = [\delta_1^T, \delta_2^T, \dots, \delta_n^T]^T$$

$$\delta_i = -4A_i\kappa_i n_i$$

其中， A_i 是顶点相关的面积； κ_i 是平均曲率； n_i 是法向向量。

因此求解这个线性方程组相当于移除法向部分和收缩面积，这一部分线性系统可以称为收缩限制条件。

第四步：因为拉普拉斯矩阵 L 是奇异的，因此需要额外的限制条件。把所有顶点的位置乘以不同的权重作为软限制条件。系统这一部分称为吸引限制条件部分。

第五步：整个骨骼抽取线性系统为

$$\begin{bmatrix} \mathbf{W}_L \mathbf{L} \\ \mathbf{W}_H \end{bmatrix} \mathbf{V}' = \begin{bmatrix} 0 \\ \mathbf{W}_H \mathbf{V} \end{bmatrix}$$

其中, \mathbf{W}_L 和 \mathbf{W}_H 是权重对角矩阵。两个矩阵的第 i 行分别表示为 $W_{L,i}$ 和 $W_{H,i}$ 。

第六步: 第五步的系统可以通过最小二乘法求解, 求得的解也是下面能量函数的最小值。

$$\|\mathbf{W}_L \mathbf{L} \mathbf{V}'\|^2 + \sum_i \mathbf{W}_{H,i}^2 \|v'_i - v_i\|^2$$

第七步: 上述的线性系统执行一次并不能得到最终的结果, 需要进行迭代, 每次迭代的时候, 需要更新 \mathbf{W}_L 和 \mathbf{W}_H 。

第八步: 迭代过程如下。

$$\begin{bmatrix} \mathbf{W}_L^t \mathbf{L}^t \\ \mathbf{W}_H^t \end{bmatrix} \mathbf{V}^{t+1} = \begin{bmatrix} 0 \\ \mathbf{W}_H^t \mathbf{V}^t \end{bmatrix} \text{ for } V^{t+1}$$

第九步: 让 A_i^t 和 A_i^0 表示当前顶点相关的面积, 那么权重更新公式如下。

$$\mathbf{W}_L^{t+1} = s_L \mathbf{W}_L^t$$

$$\mathbf{W}_{H,i}^{t+1} = \mathbf{W}_{H,i}^0 \sqrt{A_i^0 / A_i^t}$$

$$s_L = 2.0$$

第十步: 权重的初始值可以采用

$$\mathbf{W}_H^0 = 1.0$$

$$\mathbf{W}_L^0 = 10^{-3} \sqrt{A}$$

2. 核心代码

骨骼抽取算法的框架和光滑算法的框架一致, 都是求解线性系统。只是线性系统左边的矩阵和右边的向量不同。因此需要添加一些设置收缩限制条件、保持形状限制条件, 以及分别构建系统右边向量的上半部分和下半部分的函数。

1) 构建收缩部分限制条件

```
public override SparseMatrix BuildMatrixA()
{
    MatrixA = LaplaceManager.Instance.BuildMatrixCot(mesh);
    if (iterative > 0)
    {
        wL = sL * wL;
    }
    MatrixA.Scale(wL);
    return MatrixA;
}
```

2) 构建保持形状部分限制条件

```
public override void BuildConstraints(SparseMatrix A)
{
    for (int i = 0; i < mesh.Vertices.Count; i++)
    {
        A.AddRow();
        if (iterative > 0)
        {
            wH = Math.Sqrt(TriMeshUtil.ComputeAreaOneRing(BackUpMesh.Vertices[i])
                / TriMeshUtil.ComputeAreaOneRing(mesh.Vertices[i]));
        }
        A.AddElement(A.RowSize - 1, i, wH);
    }
    A.SortElement();
}
```

3) 构建方程组右边向量的上半部分

```
public override double[][] BuildBUpPart()
{
    double[][] b = new double[3][];
    int n = MatrixA.RowSize;
    for (int i = 0; i < 3; i++)
    {
        b[i] = new double[n];
        for (int j = 0; j < mesh.Vertices.Count; j++)
        {
            b[i][j] = 0;
        }
    }
    return b;
}
```

4) 构建方程组右边向量的下半部分

```
public override void UpdateHandle()
{
    int k = mesh.Vertices.Count;
    for (int i = 0; i < mesh.Vertices.Count; i++)
    {
        if (iterative > 0)
        {
            wH = Math.Sqrt(TriMeshUtil.ComputeAreaOneRing(BackUpMesh.Vertices[i])
                / TriMeshUtil.ComputeAreaOneRing(mesh.Vertices[i]));
        }
    }
}
```

```

    }
    b[0][k] = mesh.Vertices[i].Traits.Position.x * wH;
    b[1][k] = mesh.Vertices[i].Traits.Position.y * wH;
    b[2][k] = mesh.Vertices[i].Traits.Position.z * wH;
    k++;
}
}

```

5) 迭代直至收敛

```

public override void Deform()
{
    while(TriMeshUtil.ComputeVolume(mesh) >
        ConfigLaplace.Instance.VolumeThreshold)
    {
        BuildSolver();
        UpdateHandle();
        ComputeDeform();
        UpdateMesh();
        iterative++;
    }
}

```

7.5.3 骨骼抽取效果

骨骼抽取算法对于各种形状和各种拓扑的三维模型都可以成功抽取骨骼。

1. 实验一

圆环、有边界的汽车等三维模型经过骨骼提取算法后可以很成功地得到骨骼，如图7-26所示。例如，圆环提取骨骼后得到一个正圆形状。

2. 实验二

更复杂的三维模型，如狗、机械部件、扭曲的圆环等都可以成功地得到骨骼，如图7-27所示。

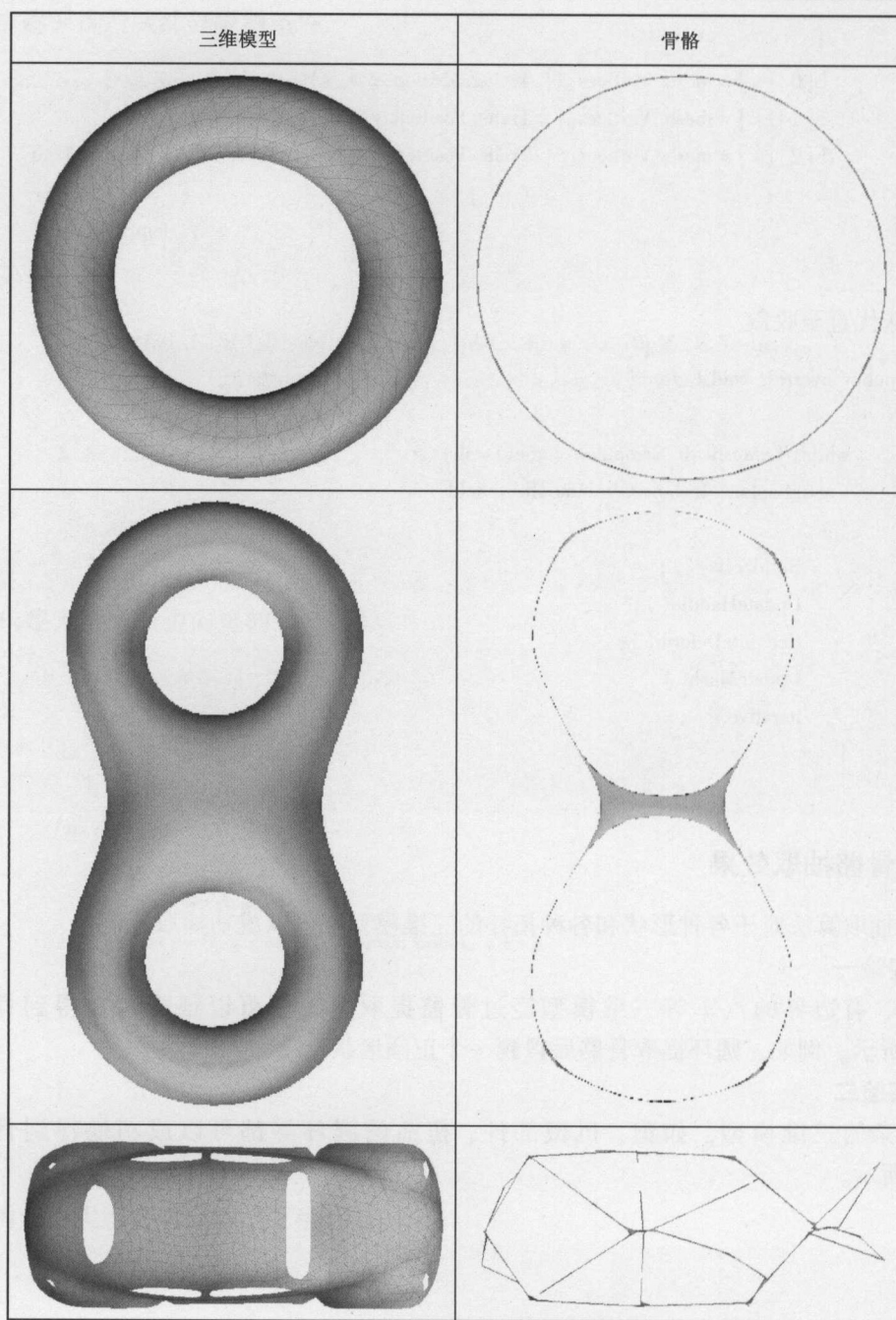


图 7-26 不同拓扑的三维模型骨骼抽取

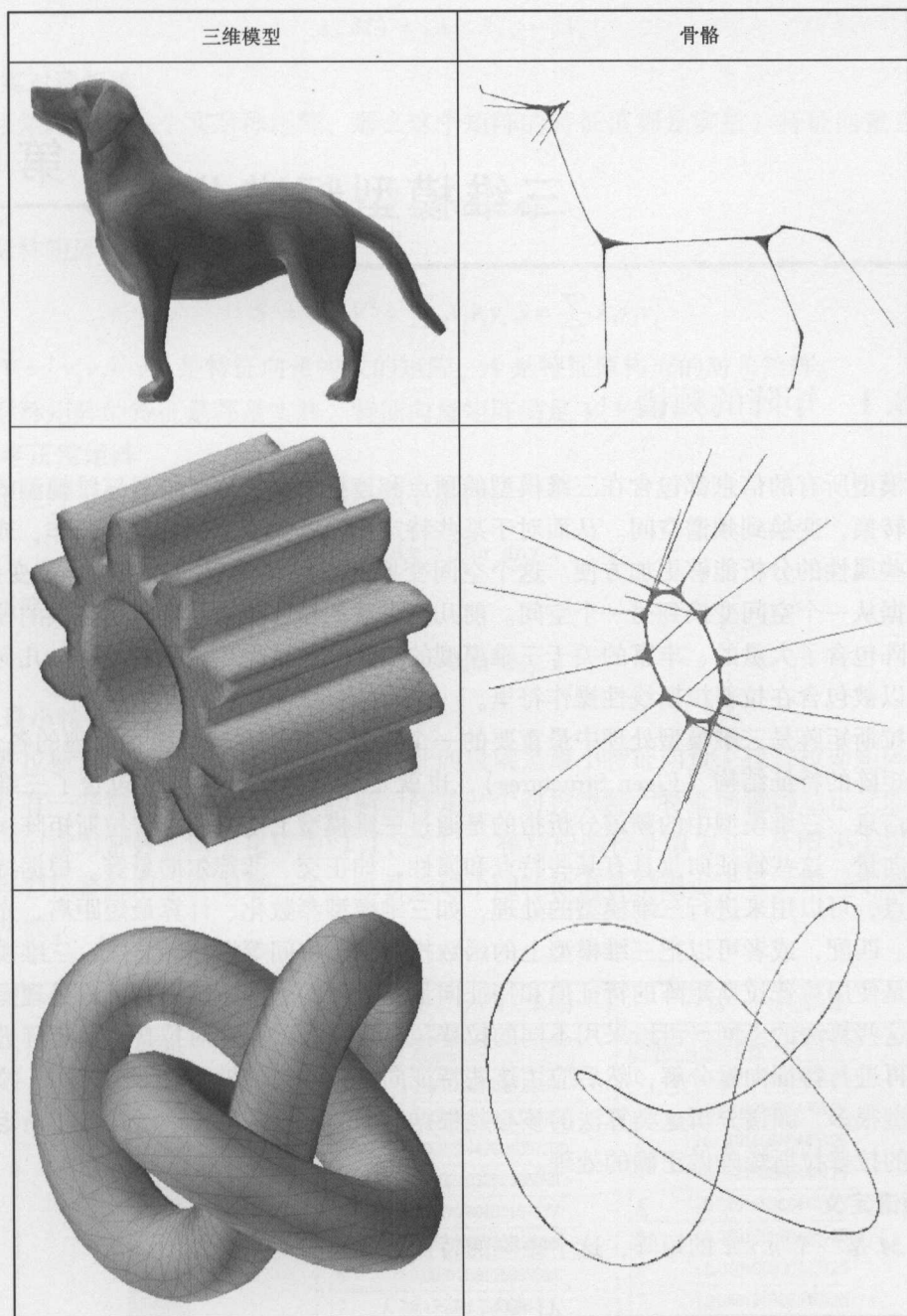


图 7-27 复杂三维模型的骨骼提取



8.1 矩阵的频谱

三维模型所有的信息都包含在三维模型的顶点和连接关系里面，但是三维模型的信息还可以经过转换，变换到频谱空间。从而对于某些特定的三维模型上的处理和操作，或者三维模型上某些属性的分析能够更加方便。这个空间变换的思路和傅里叶变换、小波变换类似，都是把数据从一个空间变换到另一个空间。前几章讲述的拉普拉斯矩阵就是变换的核心。拉普拉斯矩阵包含了大量的、丰富的关于三维模型的信息。三维模型内涵的特征、几何、模型信息都可以被包含在拉普拉斯线性操作符里。

拉普拉斯矩阵是三维模型处理中最重要的一个元素，应用在三维模型处理的各个方面。拉普拉斯矩阵的特征结构 (Eigen Structures)，也就是特征向量和特征值包含了三维模型的各种特征信息。三维模型中的频谱分析指的是通过三维模型上定义的拉普拉斯矩阵来得到相应的特征向量。这些特征向量具有某些特点和属性，如正交、菲德尔向量等。根据特征向量的这些特点，可以用来进行三维模型的处理，如三维模型参数化、计算最短距离、光滑、分段、压缩、匹配，或者可以把三维模型上的函数投影到子空间等操作。总之，三维模型的频谱分析就是使用拉普拉斯矩阵的特征值和特征向量来设计的一些三维模型上的处理操作和算法。根据这些算法的不同，可以采用不同的拉普拉斯矩阵，或者在对拉普拉斯矩阵进行一些变动后，再进行特征向量分解，然后应用这些特征向量设计三维模型处理的算法。拉普拉斯矩阵的类型很多，频谱分析这类算法的核心是根据应用的不同采用正确的拉普拉斯矩阵，或者对相应的拉普拉斯矩阵做正确的处理。

1. 频谱定义

假设 M 是一个 $n \times n$ 的矩阵，这个矩阵的特征值为

$$\lambda_1 \leq \lambda_2 \leq \dots \leq \lambda_n$$

特征向量为

$$v_1, v_2, \dots, v_n$$

也就是：

$$Mv_i = \lambda_i v_i \text{ and } v_i \neq 0, \text{ for } i \in \{1, \dots, n\}$$

$$Mv_i = \lambda_i v_i \text{ and } v_i \neq 0, \text{ for } i \in \{1, \dots, n\}$$

特征值的集合称为矩阵的频谱 (Spectrum)，即

$\lambda(M) = \{\lambda_1, \lambda_2, \dots, \lambda_n\}$

2. 实对称矩阵

如果矩阵 S 是一个实对称矩阵，那么这个矩阵的特征值都是实数，特征向量互相正交。也就是

$v_i^T v_j = 0 \text{ for } i \neq j$

那么实对称矩阵 S 可以表示为

$S = V \Lambda V^T = \sum_{i=1}^n \lambda_i v_i v_i^T$

其中， $V = [v_1 v_2 \dots v_n]$ 是特征向量构成的矩阵， Λ 是特征值构成的对角矩阵。

实对称矩阵的特征是都是实数，特征向量矩阵满足 $V^T V = I$

3. 半正定矩阵

正定矩阵指的是一个矩阵 A ，满足如下条件：

$x^T A x > 0 \text{ for any } x$

半正定矩阵指的是满足如下条件：

$x^T A x \geq 0 \text{ for any } x$

4. 最小特征向量

拉普拉斯矩阵的最小特征值对应的特征向量就是最小特征向量。拉普拉斯矩阵的最小特征向量具有一些特殊的特点。拉普拉斯矩阵是一个对称矩阵，特征向量都是正交的。它的第一个特征向量为常数，归一化后为 $(1, 1, \dots, 1)$ ，相对应的特征值为 0。从图 8-1 可以看出两个三维模型拉普拉斯矩阵的第一个，也就是最小的特征值都是逼近于零，第一个特征向量都是一个常数。

图形	特征值（从小到大排列）	第一个特征向量																																																																
	<table><tr><th>Index</th><th>EigenValue</th></tr><tr><td>0</td><td>-2.36370063307289E-17</td></tr><tr><td>1</td><td>0.00750188921105207</td></tr><tr><td>2</td><td>0.00922442604057239</td></tr><tr><td>3</td><td>0.0182403252365095</td></tr><tr><td>4</td><td>0.0434788403151177</td></tr><tr><td>5</td><td>0.0721885838604585</td></tr><tr><td>6</td><td>0.0901259195880686</td></tr><tr><td>7</td><td>0.13001473438613</td></tr><tr><td>8</td><td>0.16671651552558</td></tr><tr><td>9</td><td>0.185557961506596</td></tr><tr><td>10</td><td>0.20401059186049</td></tr><tr><td>11</td><td>0.218713637333893</td></tr><tr><td>12</td><td>0.249845085084581</td></tr><tr><td>13</td><td>0.264009635182409</td></tr><tr><td>14</td><td>0.279212283350793</td></tr></table>	Index	EigenValue	0	-2.36370063307289E-17	1	0.00750188921105207	2	0.00922442604057239	3	0.0182403252365095	4	0.0434788403151177	5	0.0721885838604585	6	0.0901259195880686	7	0.13001473438613	8	0.16671651552558	9	0.185557961506596	10	0.20401059186049	11	0.218713637333893	12	0.249845085084581	13	0.264009635182409	14	0.279212283350793	<table><tr><th>Index</th><th>EigenVector</th></tr><tr><td>0</td><td>0.040160966445125</td></tr><tr><td>1</td><td>0.040160966445125</td></tr><tr><td>2</td><td>0.040160966445125</td></tr><tr><td>3</td><td>0.040160966445125</td></tr><tr><td>4</td><td>0.040160966445125</td></tr><tr><td>5</td><td>0.040160966445125</td></tr><tr><td>6</td><td>0.040160966445125</td></tr><tr><td>7</td><td>0.040160966445125</td></tr><tr><td>8</td><td>0.040160966445125</td></tr><tr><td>9</td><td>0.040160966445125</td></tr><tr><td>10</td><td>0.040160966445125</td></tr><tr><td>11</td><td>0.040160966445125</td></tr><tr><td>12</td><td>0.040160966445125</td></tr><tr><td>13</td><td>0.040160966445125</td></tr><tr><td>14</td><td>0.040160966445125</td></tr></table>	Index	EigenVector	0	0.040160966445125	1	0.040160966445125	2	0.040160966445125	3	0.040160966445125	4	0.040160966445125	5	0.040160966445125	6	0.040160966445125	7	0.040160966445125	8	0.040160966445125	9	0.040160966445125	10	0.040160966445125	11	0.040160966445125	12	0.040160966445125	13	0.040160966445125	14	0.040160966445125
	Index	EigenValue																																																																
	0	-2.36370063307289E-17																																																																
	1	0.00750188921105207																																																																
	2	0.00922442604057239																																																																
	3	0.0182403252365095																																																																
	4	0.0434788403151177																																																																
	5	0.0721885838604585																																																																
	6	0.0901259195880686																																																																
	7	0.13001473438613																																																																
	8	0.16671651552558																																																																
	9	0.185557961506596																																																																
	10	0.20401059186049																																																																
	11	0.218713637333893																																																																
	12	0.249845085084581																																																																
13	0.264009635182409																																																																	
14	0.279212283350793																																																																	
Index	EigenVector																																																																	
0	0.040160966445125																																																																	
1	0.040160966445125																																																																	
2	0.040160966445125																																																																	
3	0.040160966445125																																																																	
4	0.040160966445125																																																																	
5	0.040160966445125																																																																	
6	0.040160966445125																																																																	
7	0.040160966445125																																																																	
8	0.040160966445125																																																																	
9	0.040160966445125																																																																	
10	0.040160966445125																																																																	
11	0.040160966445125																																																																	
12	0.040160966445125																																																																	
13	0.040160966445125																																																																	
14	0.040160966445125																																																																	

图 8-1 最小特征向量

图形	特征值（从小到大排列）		第一个特征向量	
	Index	EigenValue	Index	EigenVector
	0	1.60567612830331E-17	0	0.0199880107892113
	1	0.00388690461456412	1	0.0199880107892114
	2	0.0098892278955866	2	0.0199880107892114
	3	0.0106479242797669	3	0.0199880107892113
	4	0.0132872685398664	4	0.0199880107892113
	5	0.0150529032831064	5	0.0199880107892114
	6	0.0224262024332275	6	0.0199880107892113
	7	0.0332758443012356	7	0.0199880107892113
	8	0.0362449830552245	8	0.0199880107892113
	9	0.0399232987211092	9	0.0199880107892114
	10	0.0400831972602924	10	0.0199880107892113
	11	0.0512024767133542	11	0.0199880107892114
	12	0.0521589730655433	12	0.0199880107892114
	13	0.0562768156379139	13	0.0199880107892113
	14	0.0695693224584775	14	0.0199880107892114

图 8-1 最小特征向量（续）



8.2 菲德尔向量

拉普拉斯矩阵的第二个非零特征值对应的特征向量称为菲德尔（Fiedler）向量。菲德尔向量也是一个特殊的向量，具有很多的应用。例如，菲德尔向量可以用于三维模型的分段上面。如果一个三维模型有 V 个顶点，那么菲德尔向量是一个大小为 V 的向量。这个向量的每个元素可以对应于每个顶点上的一个数值。如果把这个数值按照等值线进行显示，那么称为菲德尔向量的等值线（Iso - Contour）。菲德尔向量的等值线具有特殊的结构和明显的规律。而一般向量的等值线都是杂乱无章的。例如，如图 8-2 所示的菲德尔向量的等值线是一圈

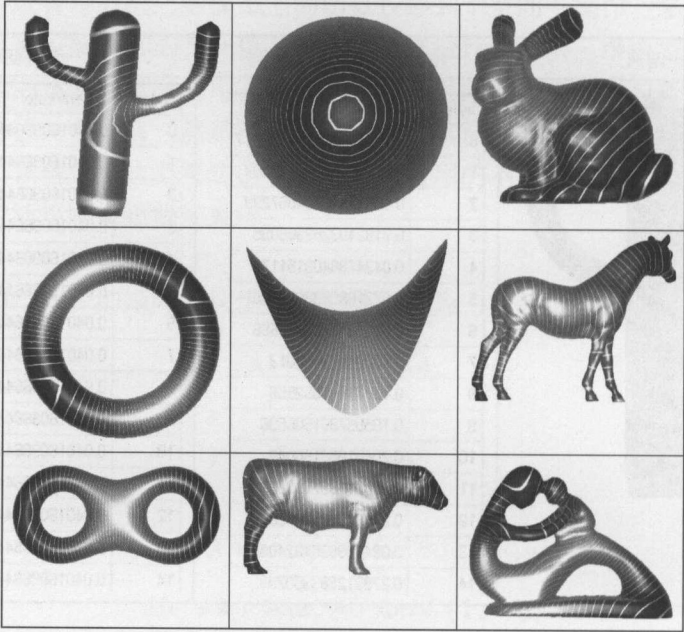


图 8-2 各种三维模型上的菲德尔向量

一圈的圆。每一个白色的线条上的顶点数值都相同。红色部分表示菲德尔向量元素值比较大的地方，蓝色部分表示菲德尔向量元素值比较小的地方。菲德尔向量元素值在三维模型上是渐变的，从红色渐变到蓝色，没有突然的中断。

在图 8-2 中，各种形状、各种拓扑的三维模型上的菲德尔向量都呈现出渐变的规律，并且每个等值线在三维模型上的间距分布是均匀的。

菲德尔向量不仅仅在封闭的三维模型上具有规律，而且在有边界的三维模型上也有规律。例如，如图 8-4 所示汽车三维模型上的菲德尔向量。拉普拉斯矩阵的菲德尔向量可以展现出渐变的、等值线的规律。拉普拉斯矩阵的第三个、第四个等和菲德尔向量相邻的特征向量也展现出类似的规律。但是后面的特征向量等值线渐渐趋于杂乱。越往后的特征向量对应的等值线就越来越失去菲德尔向量所展示的规律。如图 8-4 所示汽车三维模型的第二个（菲德尔向量）、第三个、第四个、第五个特征向量的等值线。

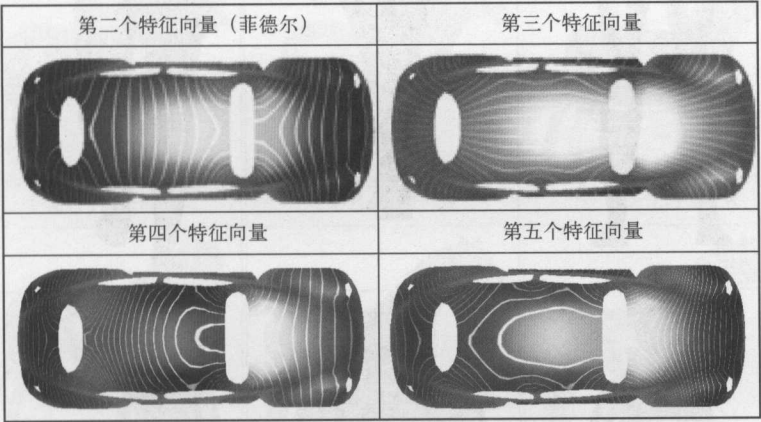


图 8-3 有边界三维模型上的特征向量

对于同一个三维模型来说，具有不同的拉普拉斯矩阵。在不规则的三维模型上，余切拉普拉斯比组合拉普拉斯得到更好的效果，如图 8-4 所示，组合的拉普拉斯矩阵得到的菲德尔向量等值线比较杂乱，而余切拉普拉斯矩阵能够得到比较光滑的特征向量。

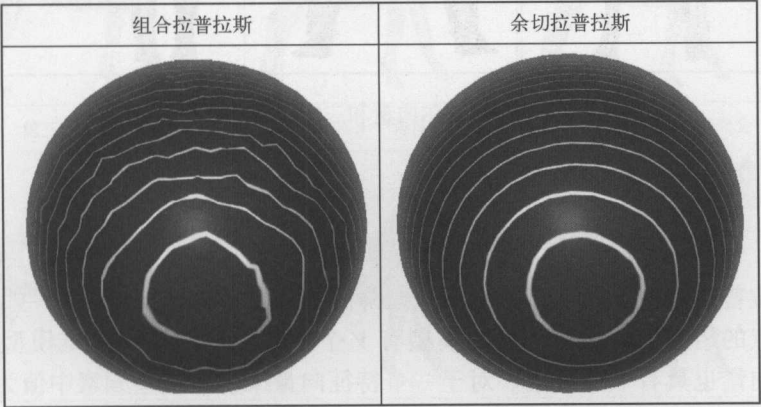


图 8-4 同一个模型不同拉普拉斯矩阵的菲德尔向量

拉普拉斯矩阵的特征向量等值线能够显示三维模型的形状，如图 8-5 所示，同一个模型的不同姿势，等值线都是能够显示身体的各个不同部分。或者模型虽然不同，但是也可以在相应的地方有类似的等值线，因此可以用于模型匹配和模型分段上。

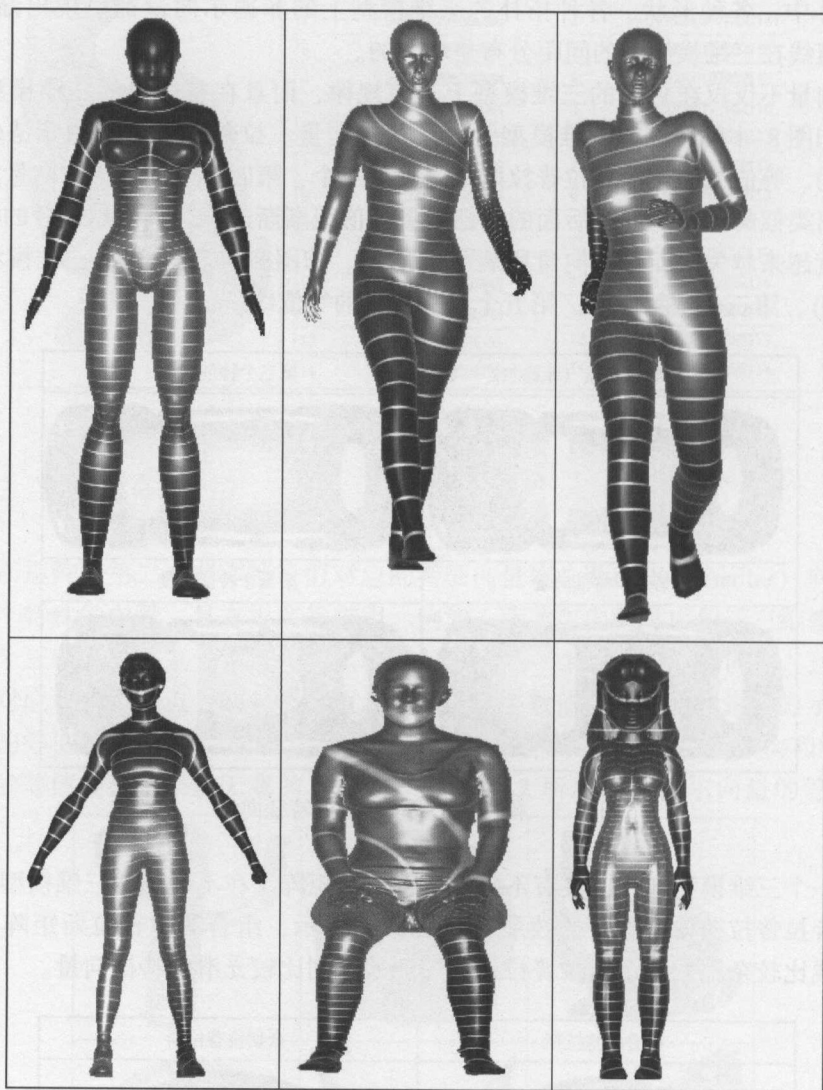


图 8-5 菲德尔向量和三维模型的形状



8.3 节点域

三维模型拉普拉斯矩阵除了菲德尔向量之外的其他特征向量，也具有一些可以描述三维模型信息和特点的结构。如果一个三维模型有 V 个顶点，那么这个三维模型有 V 个特征向量，每个特征向量也具有 V 个元素。对于一个特征向量来说，这些元素中值为零的元素的集合称为节点集合（Nodal Set）。这些值为零的元素是分段连接的，这些节点集合把整个三维模型分割为几个区域，称为节点域（Nodal domains）。

关于节点区域有如下定理 (Courant's Nodal Domain Theorem):

如果把拉普拉斯矩阵的特征值从小到大排列, 那么第 i 个特征向量最多有 i 个节点区域。这个定理仅仅指出每个特征向量节点区域的上限, 并不能得到节点区域具体的数值。

在图 8-6 中, 显示了同一个三维模型拉普拉斯矩阵不同特征向量的节点域。在节点域图中, 黑色部分的特征向量元素值为 0。红色部分表示特征向量对应的元素值比较大, 蓝色部分表示对应的元素值比较小, 从中可以看出特征向量元素值是渐变的。而黑色部分把整个模型分为互不相连的几段。随着特征向量的排序往后, 三维模型上被黑色区域分开的段越多, 如第一个特征向量对应的节点域没有分段、第二个特征向量对应的节点域有两个分段、第九个特征向量节点域有 5 个黑色分隔带。

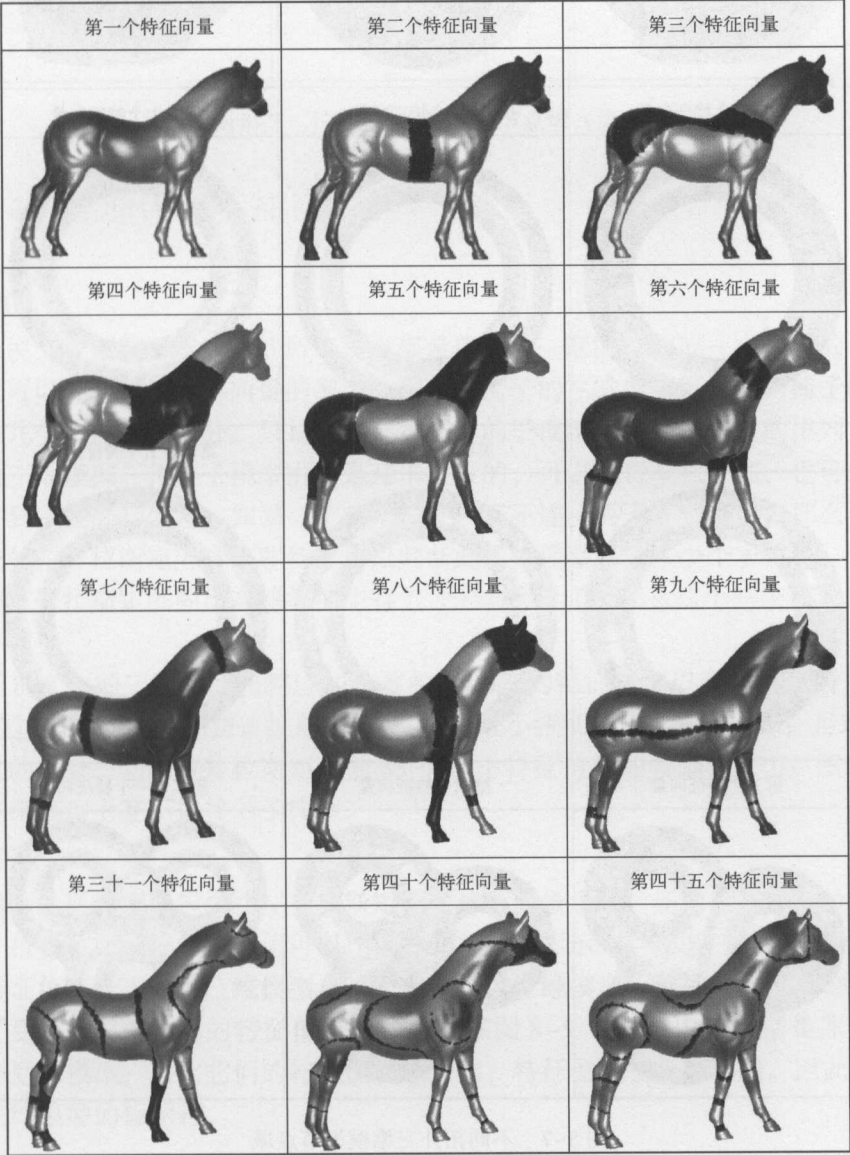


图 8-6 特征向量节点域

在不同拓扑的三维模型上，拉普拉斯矩阵特征向量的节点域也把三维模型分割为几段。例如，如图 8-7 所示是两个三维模型球形和圆环的节点域。在圆环上，由于圆环是对称的、因此节点区域呈现对称、规律的分布。

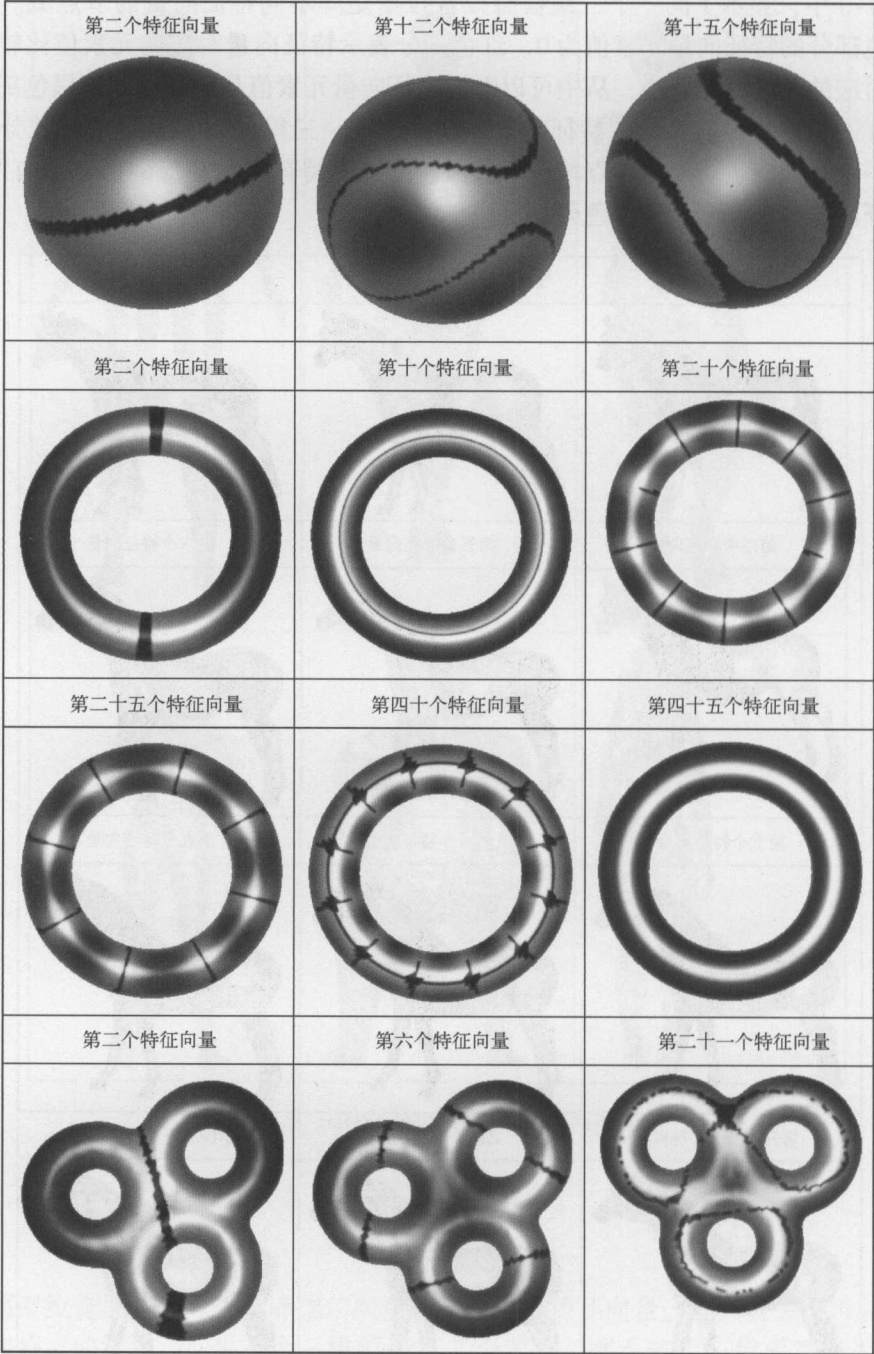


图 8-7 不同拓扑三维模型节点域

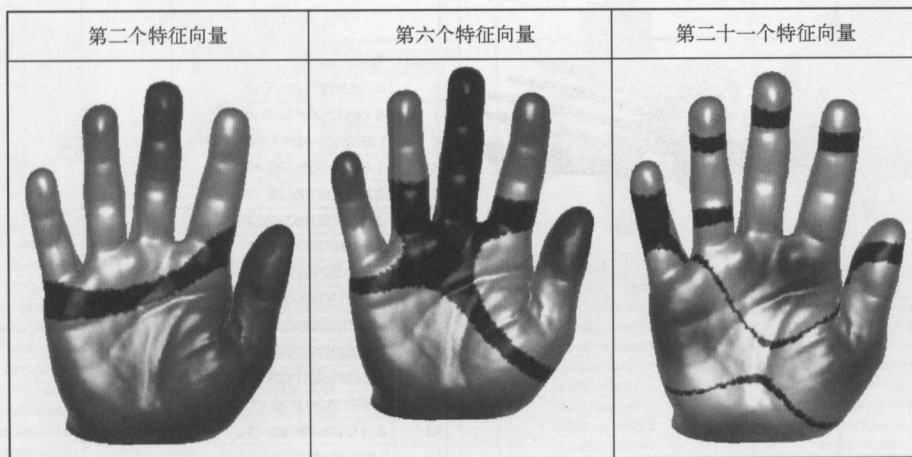


图 8-7 不同拓扑三维模型节点域 (续)



8.4 连通体和特征符

1. 连通体

通常来说,大多数三维模型的所有顶点都是连接在一起的,构成一个整体的模型。但是三维模型也可以分为几个互不向量的部分。例如,椅子的三维模型可以分为椅子的腿部和椅子的背部等几个不连接的部分。只是这些互不连接的子模型在空间位置上的排列正好能够形成一个大的三维模型。每个子模型由于是互不连接的,可以进行单独处理,也可以把所有的子模型合并到一块进行处理。如果一个三维模型由互不连接的子模型构成,那么这个大的三维模型对应的拉普拉斯矩阵有一些特征可以判断此三维模型具有多少个子模型。假如在这个三维模型的拉普拉斯矩阵的所有特征值中有 C 个特征值为 0,那么这个三维模型具有 C 个互不连接的子模型。

对比图 8-8 中的三维模型和相应的拉普拉斯矩阵的特征值可以看出:手骨三维模型有 26 个互不连接的部分,它的拉普拉斯矩阵前 26 个最小特征值非常逼近于 0;足球模型有 33 个互不连接的部分,它的拉普拉斯矩阵前 33 个最小特征值为非常逼近于 0;图左中的不同颜色表明三维模型上互不连接的子模型。

2. 特征符

三维模型的属性可以包含在拉普拉斯矩阵的特征向量里面,还内蕴在拉普拉斯矩阵的特征值里面。拉普拉斯矩阵的特征值可以作为三维模型的标示符,用于三维模型的查找。也就是可以用特征值区分不同的三维模型,唯一确定一个三维模型。对于同一个模型,如果只是姿势发生了变化,那么它们的特征值是一样的。在图 8-9 中,左边的模型是原始模型,右边的是变形后的模型,对比它们的特征值可以看出,特征值几乎是一样的。因此特征值可以作为一个三维模型的标示符。

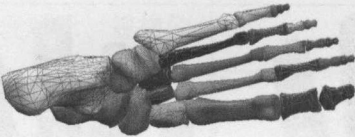

图形	特征值（从小到大排列）																																																										
	<table><tr><th>Index</th><th>EigenValue</th></tr><tr><td>0</td><td>-3.82835525732812E-18</td></tr><tr><td>1</td><td>-6.43607550507337E-18</td></tr><tr><td>2</td><td>1.28721510101468E-17</td></tr><tr><td>3</td><td>-1.46403036214308E-17</td></tr><tr><td>4</td><td>-2.73285667600038E-17</td></tr><tr><td>5</td><td>-2.77555756156289E-17</td></tr><tr><td>6</td><td>2.96059473233375E-17</td></tr><tr><td>7</td><td>3.55271367880049E-17</td></tr><tr><td>8</td><td>-3.5527136788005E-17</td></tr><tr><td>9</td><td>3.5527136788005E-17</td></tr><tr><td>10</td><td>-3.58859967555603E-17</td></tr><tr><td>11</td><td>-5.26885503211932E-17</td></tr><tr><td>12</td><td>5.75671197953788E-17</td></tr><tr><td>13</td><td>-6.21724893790088E-17</td></tr><tr><td>14</td><td>6.41790054102256E-17</td></tr><tr><td>15</td><td>-6.83214169000098E-17</td></tr><tr><td>16</td><td>-6.93889390390722E-17</td></tr><tr><td>17</td><td>9.02056207507945E-17</td></tr><tr><td>18</td><td>-9.55635006538108E-17</td></tr><tr><td>19</td><td>-1.13869028166682E-16</td></tr><tr><td>20</td><td>-1.14491749414469E-16</td></tr><tr><td>21</td><td>1.22285434596395E-16</td></tr><tr><td>22</td><td>1.27305573490351E-16</td></tr><tr><td>23</td><td>1.72701359386135E-16</td></tr><tr><td>24</td><td>-2.03012210217171E-16</td></tr><tr><td>25</td><td>2.48689957516038E-16</td></tr><tr><td>26</td><td>0.0265986336529016</td></tr><tr><td>27</td><td>0.0278885662670491</td></tr></table>	Index	EigenValue	0	-3.82835525732812E-18	1	-6.43607550507337E-18	2	1.28721510101468E-17	3	-1.46403036214308E-17	4	-2.73285667600038E-17	5	-2.77555756156289E-17	6	2.96059473233375E-17	7	3.55271367880049E-17	8	-3.5527136788005E-17	9	3.5527136788005E-17	10	-3.58859967555603E-17	11	-5.26885503211932E-17	12	5.75671197953788E-17	13	-6.21724893790088E-17	14	6.41790054102256E-17	15	-6.83214169000098E-17	16	-6.93889390390722E-17	17	9.02056207507945E-17	18	-9.55635006538108E-17	19	-1.13869028166682E-16	20	-1.14491749414469E-16	21	1.22285434596395E-16	22	1.27305573490351E-16	23	1.72701359386135E-16	24	-2.03012210217171E-16	25	2.48689957516038E-16	26	0.0265986336529016	27	0.0278885662670491
Index	EigenValue																																																										
0	-3.82835525732812E-18																																																										
1	-6.43607550507337E-18																																																										
2	1.28721510101468E-17																																																										
3	-1.46403036214308E-17																																																										
4	-2.73285667600038E-17																																																										
5	-2.77555756156289E-17																																																										
6	2.96059473233375E-17																																																										
7	3.55271367880049E-17																																																										
8	-3.5527136788005E-17																																																										
9	3.5527136788005E-17																																																										
10	-3.58859967555603E-17																																																										
11	-5.26885503211932E-17																																																										
12	5.75671197953788E-17																																																										
13	-6.21724893790088E-17																																																										
14	6.41790054102256E-17																																																										
15	-6.83214169000098E-17																																																										
16	-6.93889390390722E-17																																																										
17	9.02056207507945E-17																																																										
18	-9.55635006538108E-17																																																										
19	-1.13869028166682E-16																																																										
20	-1.14491749414469E-16																																																										
21	1.22285434596395E-16																																																										
22	1.27305573490351E-16																																																										
23	1.72701359386135E-16																																																										
24	-2.03012210217171E-16																																																										
25	2.48689957516038E-16																																																										
26	0.0265986336529016																																																										
27	0.0278885662670491																																																										
	<table><tr><th>Index</th><th>EigenValue</th></tr><tr><td>0</td><td>-1.98845914858235E-17</td></tr><tr><td>1</td><td>-3.17206578464333E-17</td></tr><tr><td>2</td><td>-3.97691829716478E-17</td></tr><tr><td>3</td><td>5.30255772955301E-17</td></tr><tr><td>4</td><td>-5.30255772955306E-17</td></tr><tr><td>5</td><td>-6.34413156928649E-17</td></tr><tr><td>6</td><td>-6.3441315692866E-17</td></tr><tr><td>7</td><td>-7.29101687813532E-17</td></tr><tr><td>8</td><td>-9.27947602671784E-17</td></tr><tr><td>9</td><td>1.0605115459106E-16</td></tr><tr><td>10</td><td>-1.11022302462516E-16</td></tr><tr><td>11</td><td>1.25935746076882E-16</td></tr><tr><td>12</td><td>-1.45820337562707E-16</td></tr><tr><td>13</td><td>-1.52448534724644E-16</td></tr><tr><td>14</td><td>1.52448534724655E-16</td></tr><tr><td>15</td><td>1.58603289232166E-16</td></tr><tr><td>16</td><td>-1.78961323372414E-16</td></tr><tr><td>17</td><td>1.7896132337242E-16</td></tr><tr><td>18</td><td>-2.0618427600181E-16</td></tr><tr><td>19</td><td>2.06184276001814E-16</td></tr><tr><td>20</td><td>-2.22044604925027E-16</td></tr><tr><td>21</td><td>2.65127886477652E-16</td></tr><tr><td>22</td><td>-2.78384280801525E-16</td></tr></table>	Index	EigenValue	0	-1.98845914858235E-17	1	-3.17206578464333E-17	2	-3.97691829716478E-17	3	5.30255772955301E-17	4	-5.30255772955306E-17	5	-6.34413156928649E-17	6	-6.3441315692866E-17	7	-7.29101687813532E-17	8	-9.27947602671784E-17	9	1.0605115459106E-16	10	-1.11022302462516E-16	11	1.25935746076882E-16	12	-1.45820337562707E-16	13	-1.52448534724644E-16	14	1.52448534724655E-16	15	1.58603289232166E-16	16	-1.78961323372414E-16	17	1.7896132337242E-16	18	-2.0618427600181E-16	19	2.06184276001814E-16	20	-2.22044604925027E-16	21	2.65127886477652E-16	22	-2.78384280801525E-16										
Index	EigenValue																																																										
0	-1.98845914858235E-17																																																										
1	-3.17206578464333E-17																																																										
2	-3.97691829716478E-17																																																										
3	5.30255772955301E-17																																																										
4	-5.30255772955306E-17																																																										
5	-6.34413156928649E-17																																																										
6	-6.3441315692866E-17																																																										
7	-7.29101687813532E-17																																																										
8	-9.27947602671784E-17																																																										
9	1.0605115459106E-16																																																										
10	-1.11022302462516E-16																																																										
11	1.25935746076882E-16																																																										
12	-1.45820337562707E-16																																																										
13	-1.52448534724644E-16																																																										
14	1.52448534724655E-16																																																										
15	1.58603289232166E-16																																																										
16	-1.78961323372414E-16																																																										
17	1.7896132337242E-16																																																										
18	-2.0618427600181E-16																																																										
19	2.06184276001814E-16																																																										
20	-2.22044604925027E-16																																																										
21	2.65127886477652E-16																																																										
22	-2.78384280801525E-16																																																										

图 8-8 拉普拉斯特征值和连通体

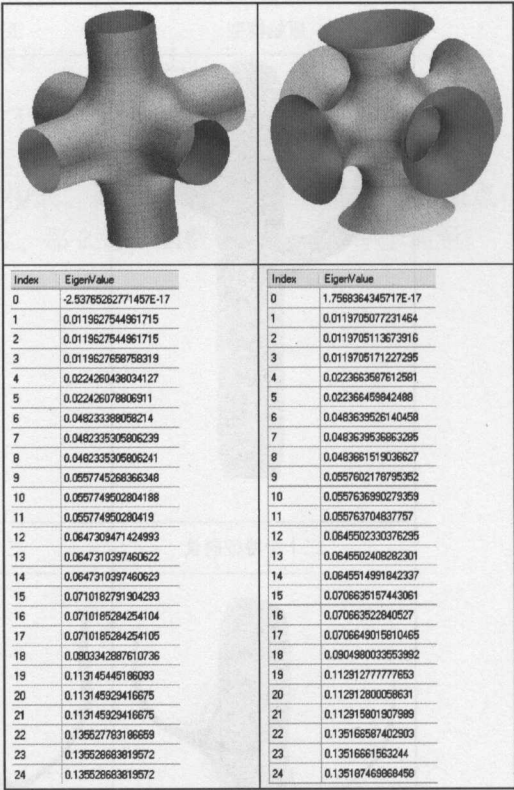
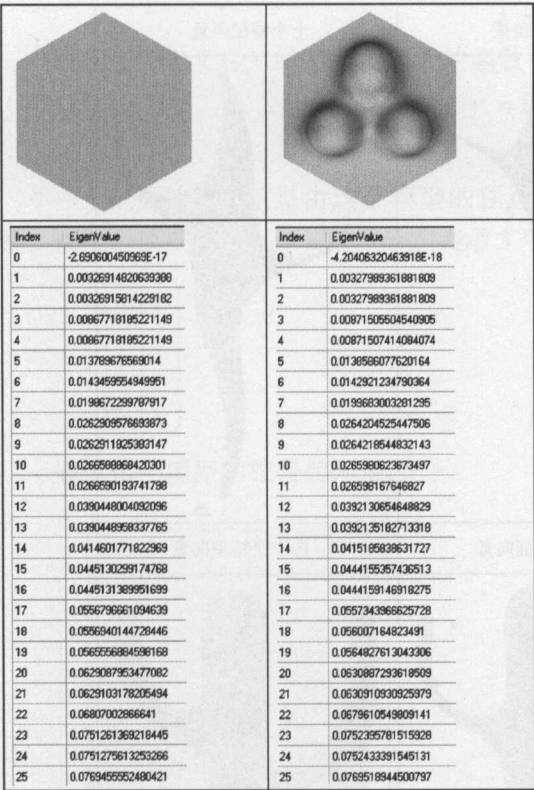


图 8-9 三维模型的特征值标示符



8.5 特征向量近似

8.5.1 数学原理

假如一个三维模型有 V 个顶点，那么这个三维模型具有 V 个特征向量。虽然所有的信息都包含在这 V 个特征向量中，但是这 V 个特征向量并不都是平等重要的，有些向量包含了更多的信息。三维模型较小的特征值对应的特征向量比较大的特征值对应的特征向量，在三维模型的属性分析上具有更重要的地位。大部分三维模型的信息都包含在较小的特征值对应的特征向量上。这些较小的特征向量可以用在三维模型近似上。例如，在图 8-10 中仙人掌三维模型有 620 个顶点，也就是有 620 个特征向量。分别用前五个、前十个、前二十个、前三十个、前四十个，最小的特征值对应的特征向量近似的效果如图 8-10 所示。从中可以看出，即使只用前五个特征向量就可以具有仙人掌的大致全局结构。采用前 40 个特征向量就可以近似出和原来三维模型差不多的仙人掌，这只占全部 620 个特征向量的很小一部分。从中可以得知三维模型的大部分信息都集中在了最小的特征值对应的特征向量上了，越往后的特征向量，对三维模型全局形状的贡献所占比例越少。

定理： S 是一个实对称矩阵，那么它的特征值 $\lambda_1 \leq \lambda_2 \leq \dots \leq \lambda_n$ 和相对应的特征向量 v_1, v_2, \dots, v_{i-1} 满足如下公式。

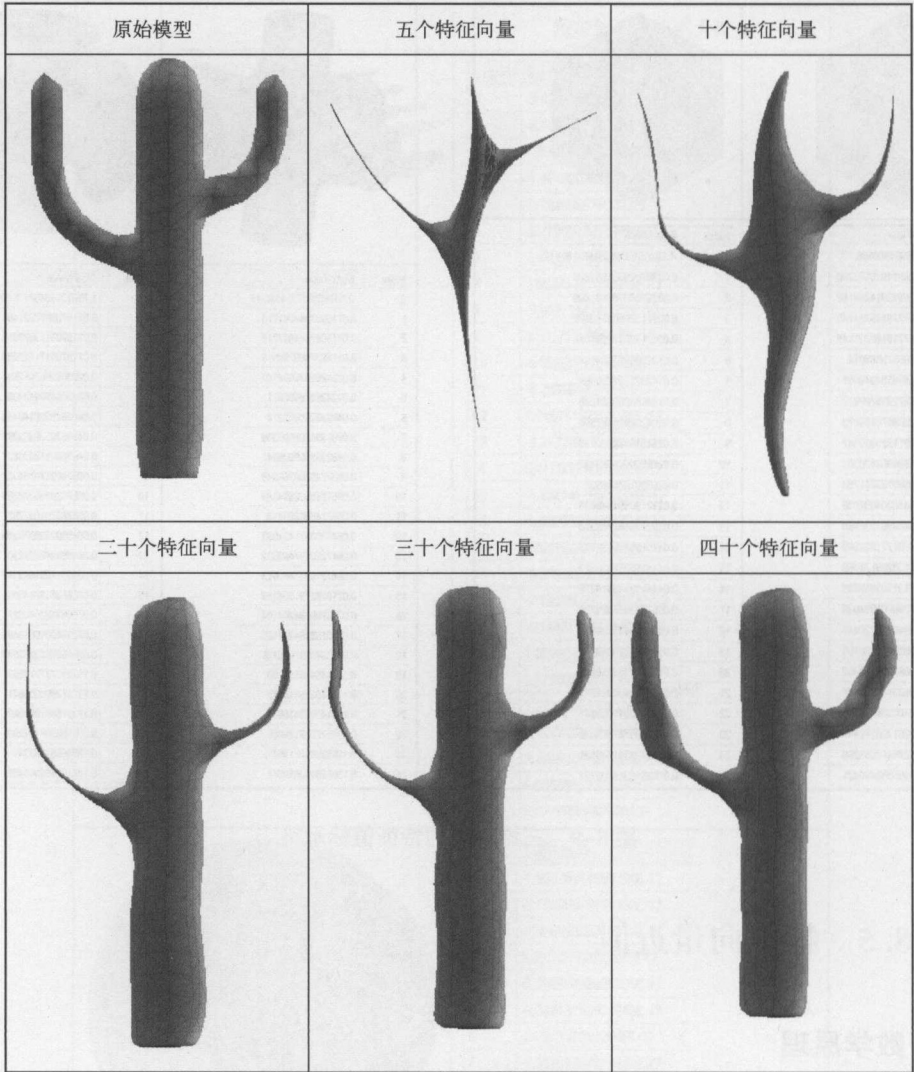


图 8-10 仙人掌三维模型特征向量近似

$$\lambda_i = \min_{\|v\|_2=1} v^T S v$$
$$v^T v_k = 0, \forall 1 \leq k \leq i-1$$

定理：假设 $\text{tr}(M)$ 表示矩阵的迹，那么

$$\sum_{i=1}^k \lambda_i = \min_{U \in R^{n \times k}} \text{tr}(U^T S U)$$

从这两个定理可以得出，给定 k 个互相正交的向量，使矩阵的迹最小的 k 个向量是这个矩阵的 k 个最小的特征向量。

三维模型上的拉普拉斯矩阵是一个线性算子，它的特征向量构成一个向量空间。三维模型上的函数可以投影在这个向量空间上。

根据 $Au = \lambda u, u \neq 0$

拉普拉斯矩阵 A 可以分解为

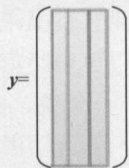
$$A = U\Lambda U^T$$

因此一个定义在三维模型顶点上的函数可以投影在子空间上:

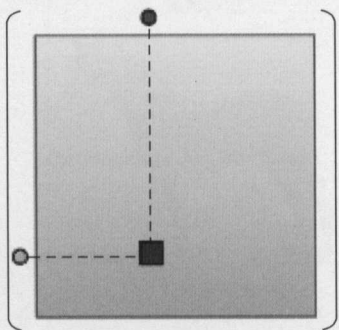
$$y' = U_{(K)} U_{(K)}^T y$$

$$\hat{y} = U^T y$$

一个模型的几何信息由这个模型的顶点位置构成。如果这个函数是三维模型的顶点位置, 也就是顶点的 x 、 y 、 z 坐标分别表示 3 个函数。那么每个函数可以表示为一个向量。



假设这个模型的拉普拉斯矩阵为



矩阵分解为:

$$A = U \Lambda U^T$$

那么位置向量在子空间的投影为

$$y' = U_{(3)} U_{(3)}^T y$$

8.5.2 近似算法步骤

采用若干个较小的特征值对应的特征向量近似原来的三维模型, 首先需要计算出较小的特征向量, 其次需要得到三维顶点位置坐标构成的 x 、 y 、 z 三个方向的向量, 然后把这 3 个向量投射到较小的特征向量所对应的向量空间上。最后再重建得到近似的模型。

第一步: 计算特征向量和特征值。

```
Eigen eigen = EigenManager.Instance.ComputeEigen(mesh,
    EnumLaplaceMatrix.LaplaceCot, reconNum + 10);
```

第二步：得到三维模型的 x 、 y 、 z 三个方向的坐标向量。

```
List<double> X = TriMeshUtil.GetX(mesh);
List<double> Y = TriMeshUtil.GetY(mesh);
List<double> Z = TriMeshUtil.GetZ(mesh);
```

第三步：把这 3 个向量投射到特征向量上。

```
List<double> factorX = new List<double>(reconNum);
List<double> factorY = new List<double>(reconNum);
List<double> factorZ = new List<double>(reconNum);
for(int i=0; i<reconNum; i++)
{
    List<double> eigenVector = eigen.GetEigenVector(i);
    double valueX = TriMeshUtil.Multiply(eigenVector, X);
    double valueY = TriMeshUtil.Multiply(eigenVector, Y);
    double valueZ = TriMeshUtil.Multiply(eigenVector, Z);
    factorX.Add(valueX);
    factorY.Add(valueY);
    factorZ.Add(valueZ);
}
```

第四步：用上一步得到的值计算新的三维模型的 x 、 y 、 z 三个方向的坐标向量。

```
double[] reconX = new double[mesh.Vertices.Count];
double[] reconY = new double[mesh.Vertices.Count];
double[] reconZ = new double[mesh.Vertices.Count];
for(int i=0; i<reconNum; i++)
{
    List<double> eigenVector = eigen.GetEigenVector(i);
    List<double> tempX = TriMeshUtil.Multiply(eigenVector, factorX[i]);
    reconX = TriMeshUtil.Add(reconX, tempX);

    List<double> tempY = TriMeshUtil.Multiply(eigenVector, factorY[i]);
    reconY = TriMeshUtil.Add(reconY, tempY);
    List<double> tempZ = TriMeshUtil.Multiply(eigenVector, factorZ[i]);
    reconZ = TriMeshUtil.Add(reconZ, tempZ);
}
```

第五步：用新的 x 、 y 、 z 坐标更新三维模型。

```
for(int i=0; i<mesh.Vertices.Count; i++)
{
```



```
mesh. Vertices[i]. Traits. Position. x = reconX[i];
mesh. Vertices[i]. Traits. Position. y = reconY[i];
mesh. Vertices[i]. Traits. Position. z = reconZ[i];
}
```

8.5.3 效果分析

采用特征向量进行近似可以成功的作用于各种形状的三维模型上，并且对于复杂拓扑的三维模型也可以很好的近似。

1. 实验一

如图 8-11 所示，这个马的三维模型有 19851 个顶点，也就是有 19851 个特征向量。

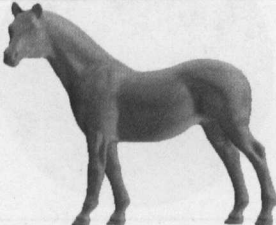
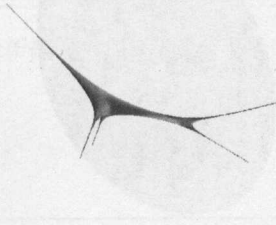
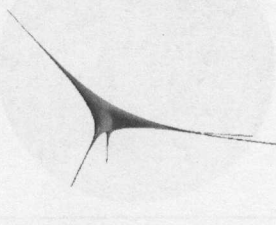
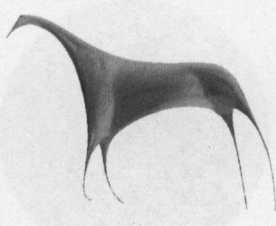
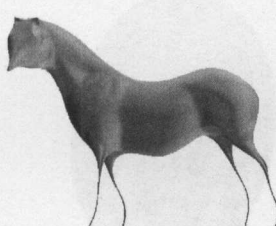
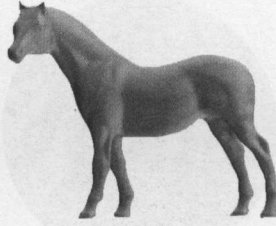
原始模型	三个特征	五个特征向量
		
十个特征向量	五十个特征向量	两百个特征向量
		

图 8-11 马的三维模型特征向量近似

2. 实验二

如图 8-12 所示，两个三维球面模型分别有 10242 个顶点和 1000 个顶点，也就是分别有 10242 个和 1000 个特征向量，从中可以看出，顶点数少的球用比较少的特征向量就可以重建。并且由于球是对称的，用很少的特征向量就可以很逼近原来的模型。假如球的顶点数量越少，用越少的特征向量就可以得到几乎一样的近似，如同样用 30 个特征向量，第一行的球比第二行的球顶点数量更多，因此得到的是一个椭圆，而第二行的球就可以得到一个近似的正圆。

3. 实验三

对于不同拓扑和有边界的三维模型，也可以用特征向量近似。例如，如图 8-13 所示的圆环、具有很多边界的汽车三维模型，都可以成功地用特征向量进行近似重建。

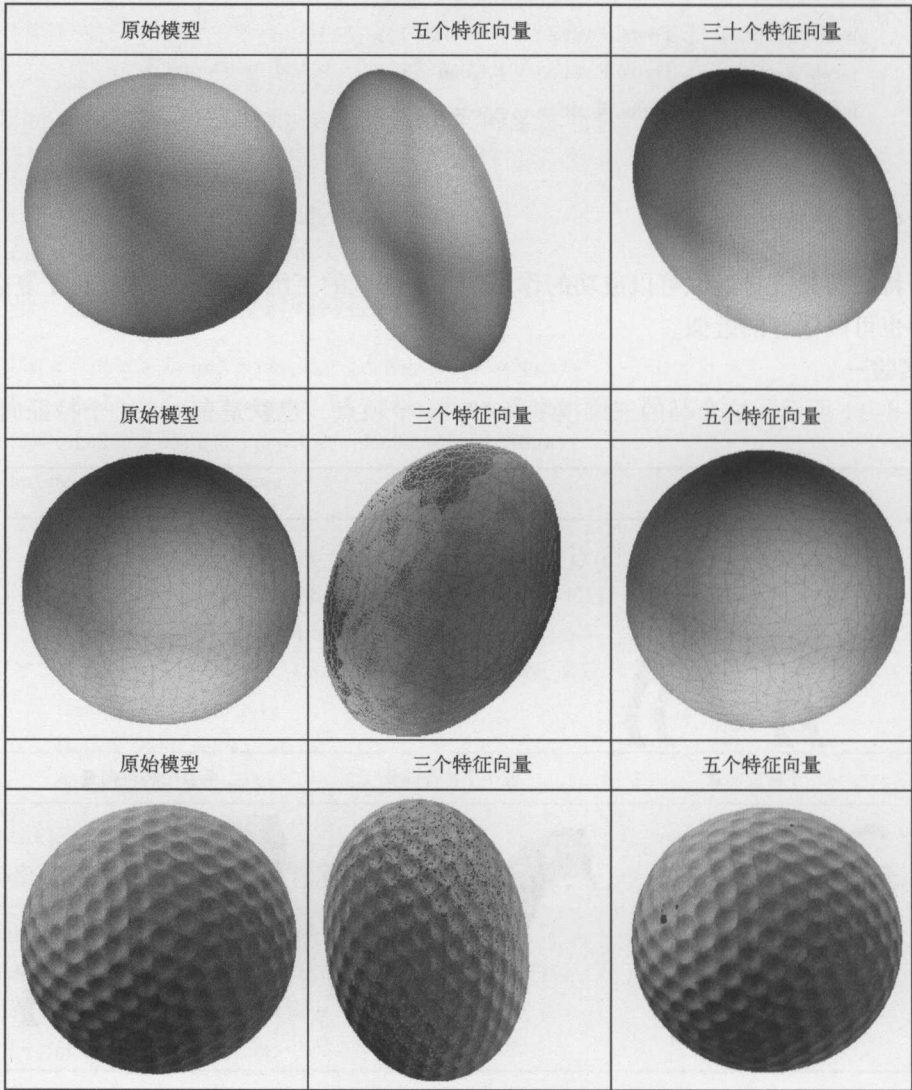


图 8-12 球三维模型特征向量近似

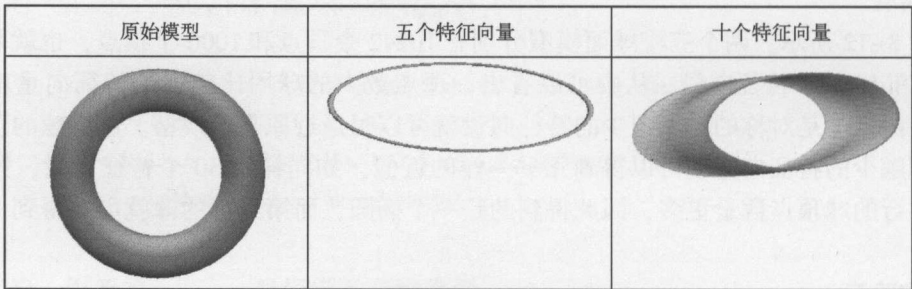


图 8-13 复杂拓扑结构的三维模型特征向量近似

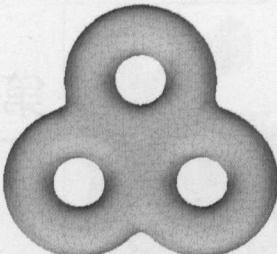
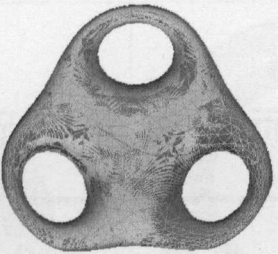
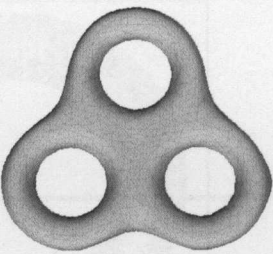
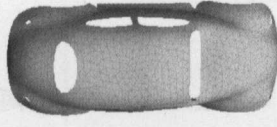
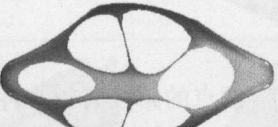
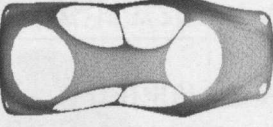
原始模型	五个特征向量	十个特征向量
		
原始模型	五个特征向量	十个特征向量
		

图 8-13 复杂拓扑结构的三维模型特征向量近似 (续)



9.1 最短距离概念

三维模型上给定任意两点，连接两点的路径有很多种，其中最短的一条称为两点间的最短距离。假如限制连接两点之间的路径必须是三维模型上的边，而不能穿过三维模型上的面，这样的最短距离，称为基于边的最短距离。假如没有这个限制，也就是路径可以穿过三角形面，那就称为基于面的最短距离。对于固定的两点，基于面的最短距离要比基于边的最短距离数值更小。计算三维模型两点之间的最短距离是三维模型一个重要的操作。两点间的最短距离可以用于三维模型分段、匹配等后续的其他高级三维模型操作。如图9-1所示，三维兔子模型上红色点和蓝色点之间的基于边的最短距离是0.54917，最短距离的路径经过三维模型上的边。而仙人掌模型上红色顶点和蓝色顶点之间的基于边的最短距离是0.61642。

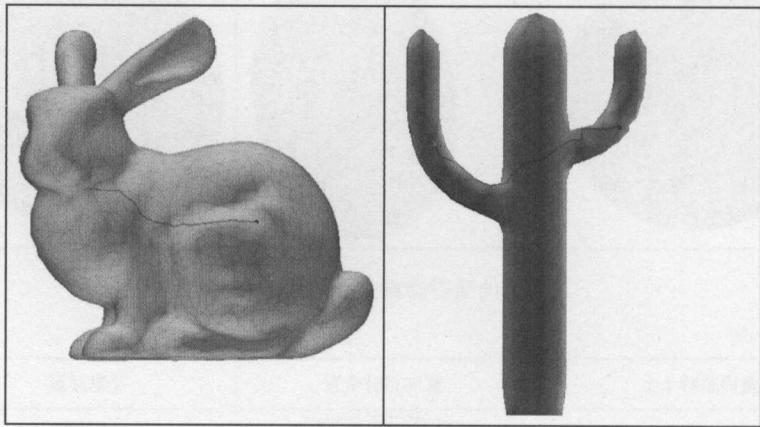


图9-1 基于边的顶点之间最短距离

基于面的最短距离可以用等值线的方法来显示。也就是设定一个顶点作为原点，三维模型上的其他顶点到这个原点的基于面的最短距离的数值可以用相应的颜色来展示。一般通常把最短距离计算出来后再归一化为0~1，然后映射到颜色。三维模型上每个点的距离随着距离源点的远近从蓝色变为红色，白色的圆圈是等值线，每个圈上的最短距离值都相等。例如，如图9-2所示的4个三维模型上基于面的最短距离，每个三维模型上白圈包围的顶点就是原点，可以看出来距离原点越远，颜色越来越红。并且可以看出白色圆圈是光滑的向远处

传递,表明最短距离随着顶点的远离而增加。

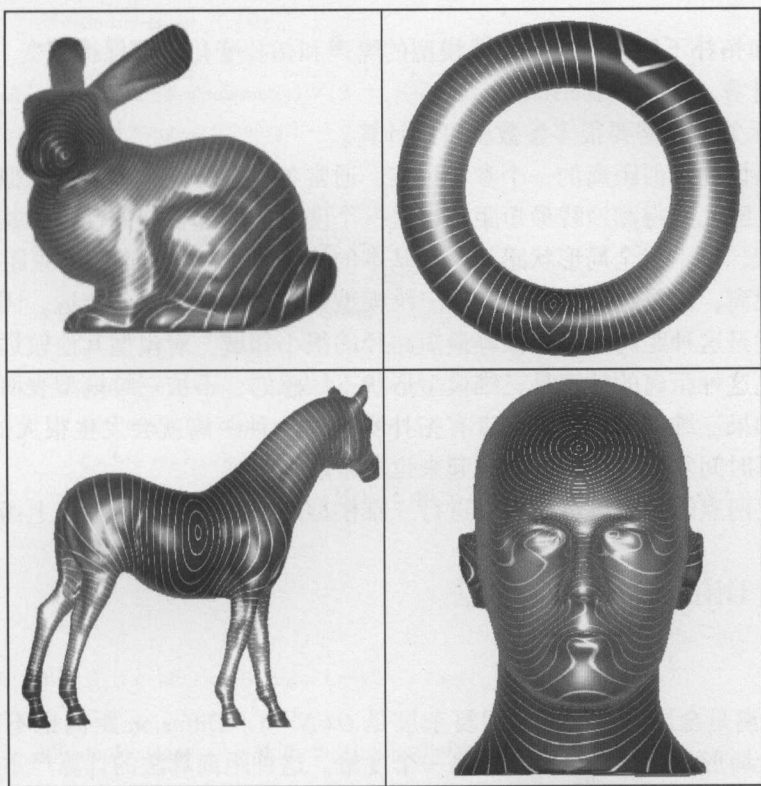


图 9-2 基于面的最短距离等值线

计算最短距离的算法有很多种,对于基于边的最短距离,最简单的方法是用迪杰斯特拉方法。很多能够准确计算最短距离的算法需要的时间很长。所以通常采用其他近似但速度快的方法来计算最短距离。采用拉普拉斯矩阵的特征向量进行距离计算是一个非常重要的求解近似距离的方法。这些算法首先分析最短距离的属性和应该满足的条件,然后根据这些属性和条件来定义一个最短距离计算的公式。这个公式可以通过特征向量很快计算出来。这样计算出来的最短距离虽然是准确值的近似,但是具有和最短距离类似的特点。最重要的不是最短距离的准确性,而是根据这些近似的最短距离,可以把应用到后续的分段等其他的操作上。

三维模型上的两点定义一个距离,这个距离是一个函数。这个距离的定义可以是欧几里得意义上测地线的最短距离,这样称为准确的最短距离,也可以是其他近似准确最短距离的定义。对于三维模型处理,一个好的距离的定义或者算法需要满足如下的条件。

- (1) 度量 (metric), 也就是非负值、对称和三角形不等性。
- (2) 渐进 (gradual): 对于目标顶点和源顶点的扰动是光滑的。
- (3) 局部各向同性 (Local Isotropic): 如果目标顶点逼近源顶点, 那么这个距离逼近于最短距离。
- (4) 感知全局形状 (global shape aware): 当目标顶点远离源顶点时候, 计算出来的近似距离能够反映三维模型的形状。

(5) 等距不变性 (isometry invariant): 计算出来的近似距离不随三维模型的等距变化而发生变化。

(6) 噪声和拓扑不敏感: 对于三维模型的噪声和拓扑变化能够保持不变。

(7) 容易计算。

(8) 参数无关: 不需要很多参数就可以计算。

这些属性是设计近似距离的一个参考标准, 通常各种方法计算出来的近似距离只满足其中的几种条件。欧几里得测地线最短距离也是一个度量, 局部各向同性, 但是不光滑, 不抗噪音和拓扑改变, 也不能全局形状感知。最基本的测地线 (Geodesic) 最短距离是最常用且最直观的最短距离, 但是这种距离对某些三维模型处理应用并不十分合适。因为这种距离是局部距离, 也就是这种距离计算只依赖最短路径的很小邻域, 对模型其他较远的地方信息数据不敏感。因此这种距离的计算是三维模型形状不敏感的, 不被三维模型表面的其他部分所影响。其次, 如果三维模型有洞, 或者有拓扑噪声, 这种距离就会发生很大的变化。最后, 这种距离的计算时间复杂度很高, 实现起来也非常麻烦。

三维模型上两点间的距离可以用来进行三维模型匹配和三维模型顶点上函数的插值。



9.2 Diffusion 距离算法

1. 定义

Diffusion 距离是全局的, 计算时间复杂度是 $O(N^{1.5})$ 。Diffusion 距离也不满足局部各向同性, 也不能全局形状感知, 也不一定是一个度量。这种距离算法的计算严重依赖于一个大于零的参数, 这个参数如果设置太大, 那么距离只和最小的一些特征值和相对应的一些特征向量有关。得到的结果具有比较好的全局属性, 但是局部属性就比较差。如果参数值设置比较小, 那么得到的局部属性比较好, 但是较远的地方计算出来的值就无法预期。计算的时候为了和三维模型的缩放无关, 通常把参数进行如下的调整:

$$t \leftarrow t / (2\lambda_1)$$

Diffusion 距离公式:

$$d_D(x, y)^2 = \sum_{k=1}^{\infty} e^{-2t\lambda_k} (\phi_k(x) - \phi_k(y))^2$$

其中, $\phi_k(x)$ 和 λ_k 是特征向量和特征值。

2. 实现代码

根据 Diffusion 距离的定义公式, 需要全部的特征值和特征向量, 但是在实现的时候, 采用部分特征值和特征向量就可以得到比较近似的结果。首先计算若干个最小的特征值和对应的特征向量, 然后可以根据上述公式进行计算。

```
public static double[] ComputeDistanceDiffusion(int originVertexIndex,
                                                double parameterT,
                                                Eigen eigens)
{
    double[] diffusionDistance = new double[eigens.EigenVectorSize];
    for(int i = 0; i < diffusionDistance.Length; i++)
```



```

    {
        diffusionDistance[i] = 0;
    }
    double optParamterT = parameterT / (2 * eigens. GetEigenValue(1));
    for(int i = 1; i < eigens. Count; i++)
    {
        EigenPair pair = eigens. SortedEigens[i];
        double eigenValue = pair. EigenValue;
        List<double> eigenVector = pair. EigenVector;
        for(int j = 0; j < eigenVector. Count; j++)
        {
            double orig = eigenVector[originVertexIndex];
            double y = eigenVector[j];
            double res = orig - y;
            diffusionDistance[j] += Math. Exp(-2 * optParamterT * eigenValue)
                                   * (res * res);
        }
    }
    for(int i = 0; i < diffusionDistance. Length; i++)
    {
        diffusionDistance[i] = Math. Sqrt(diffusionDistance[i]);
    }
    return diffusionDistance;
}

```

3. 效果图

Diffusion 距离在各种三维模型上都可以显示出等值线, 根据参数的不同, 这些等值线呈现不同的分布。

例如图 9-3 中的手掌三维模型上, 参数分别为 0.125、0.25、0.5、1.0 时分别展现不同的等值线。

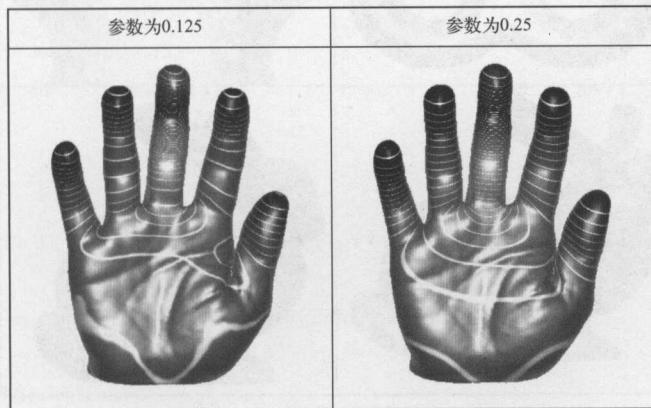


图 9-3 三维模型上不同参数的 Diffusion 距离等值线图



图 9-3 三维模型上不同参数的 Diffusion 距离等值线图（续）

不同拓扑和有边界三维模型上，也可以计算 Diffusion 距离。如图 9-4 所示，从复杂拓扑的三维模型和有边界的兔子三维模型上可以看出，Diffusion 距离可以很好地适应有洞的三维模型。

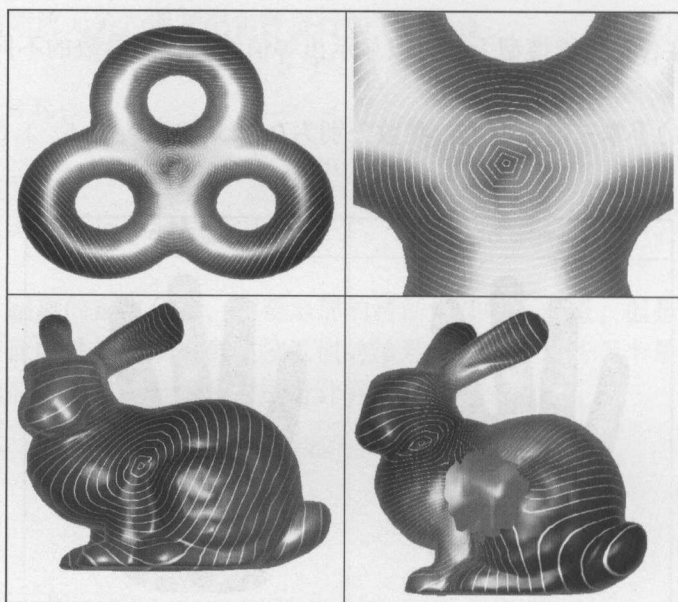


图 9-4 有洞和复杂拓扑的三维模型 Diffusion 距离等值线图



9.3 Commute Time 距离算法

根据特征值和特征向量可以定义 Diffusion 距离,也可以采用不同的公式,对这些特征值和特征向量进行不同的计算,得到其他的最短距离定义。当然,这些公式不是任意定义的,这些公式需要能够满足最短距离的各种属性。Commute Time 最短距离是另一个利用拉普拉斯矩阵的特征值和特征向量来定义和计算距离的方法。

1. Commute Time 最短距离定义公式

$$d_c(x, y)^2 = \sum_{k=1}^{\infty} \frac{1}{\lambda_k} (\phi_k(x) - \phi_k(y))^2$$

2. 实现代码

Commute Time 最短距离的实现方法和 Diffusion 类似,都是需要求得若干特征值和特征向量,只是采用不同的公式对这些特征值和特征向量进行计算。

```
public static double[] ComputeDistanceCommuteTime(int originVertexIndex,
                                                    Eigen eigens)
{
    double[] diffusionDistance = new double[eigens.EigenVectorSize];
    for(int i=0; i<diffusionDistance.Length; i++)
    {
        diffusionDistance[i] = 0;
    }
    for(int i=1; i<eigens.Count; i++)
    {
        EigenPair pair = eigens.SortedEigens[i];
        double eigenValue = pair.EigenValue;
        List<double> eigenVector = pair.EigenVector;
        for(int j=0; j<eigenVector.Count; j++)
        {
            double orig = eigenVector[originVertexIndex];
            double y = eigenVector[j];
            double res = orig - y;
            diffusionDistance[j] += (1/eigenValue) * (res * res);
        }
    }
    for(int i=0; i<diffusionDistance.Length; i++)
    {
        diffusionDistance[i] = Math.Sqrt(diffusionDistance[i]);
    }
    return diffusionDistance;
}
```


3. 效果图示

Commute Time 最短距离也可以在各种各样的三维模型上得到光滑的结果。如图 9-5 展示了马、鼓、仙人掌等三维模型上 Commute Time 最短距离的等值线。

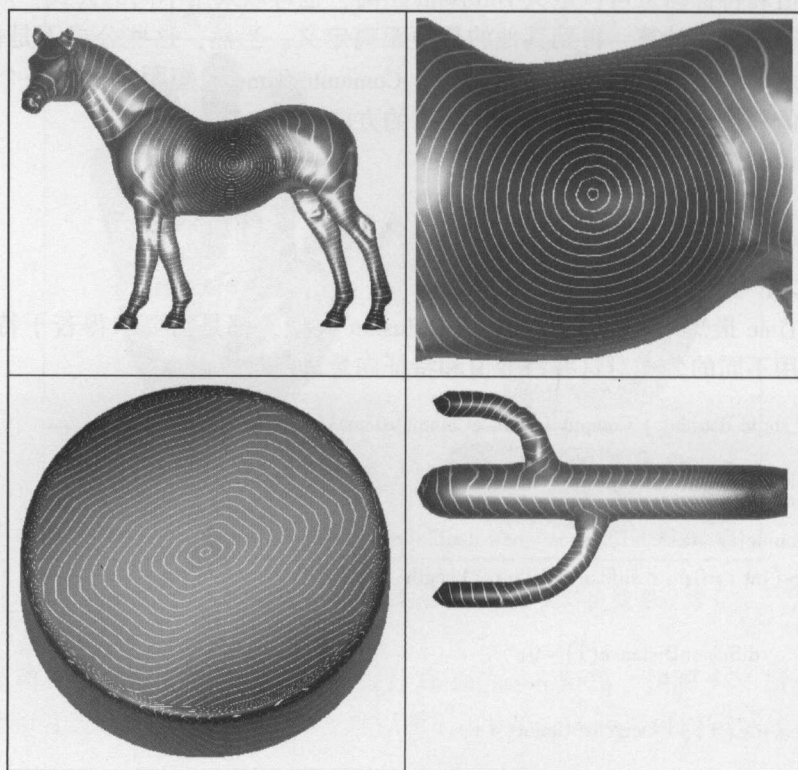


图 9-5 Commute Time 最短距离图示



9.4 双和谐距离

双和谐（Biharmonic）距离和 Diffusion 距离、Commute - time 距离类似，也是使用拉普拉斯矩阵的特征值和特征向量。但是双和谐距离的设计是针对上面两种距离的缺点进行设计的，因此能够满足第一节所说的一个良好的距离需要的各种性质。

1. 双和谐最短定义公式

$$d_B(x, y)^2 = \sum_{k=1}^{\infty} \frac{(\phi_k(x) - \phi_k(y))^2}{\lambda_k^2}$$

2. 实现代码

双和谐实现代码和上两节的一样，只是具体计算的公式有区别。

```
public static double[] ComputeDistanceBiharmonic(int sourceVertex, Eigen eigens)
{
    double[] biharmonicDistance = new double[eigens.EigenVectorSize];
```

```

for ( int i = 0; i < biharmonicDistance. Length; i ++ )
{
    biharmonicDistance[ i ] = 0;
}
for ( int i = 1; i < eigens. Count; i ++ )
{
    EigenPair pair = eigens. SortedEigens[ i ];
    double eigenValue = pair. EigenValue;
    List < double > eigenVector = pair. EigenVector;
    for ( int j = 0; j < eigenVector. Count; j ++ )
    {
        double orig = eigenVector[ sourceVertex ];
        double y = eigenVector[ j ];
        double res = orig - y;
        biharmonicDistance[ j ] += ( res * res ) / ( ( eigenValue ) * ( eigenValue ) );
    }
}
for ( int i = 0; i < biharmonicDistance. Length; i ++ )
{
    biharmonicDistance[ i ] = Math. Sqrt( biharmonicDistance[ i ] );
}
return biharmonicDistance;
}

```

3. 效果图示

双和谐距离等值线如图 9-6 所示。

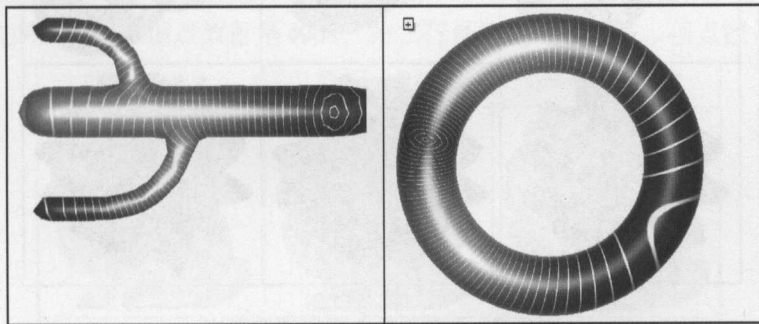


图 9-6 双和谐距离等值线

4. 距离对比

虽然 Diffusion、Commute Time、Biharmonic 都是利用拉普拉斯矩阵的特征值和特征向量进行定义和计算的。但是由于计算公式的不同，在同一个三维模型上，所表现的等值线模式和规律也不一样。图 9-7 中展示的是各种定义的最短距离在同样的模型和原点相同的情况下，最短距离等值线图对比。从中可以看出不同距离在各种情况下的特点。例如，有边界、有噪声、有拓扑改变、全局形状、局部各项同性等情况下的特点。

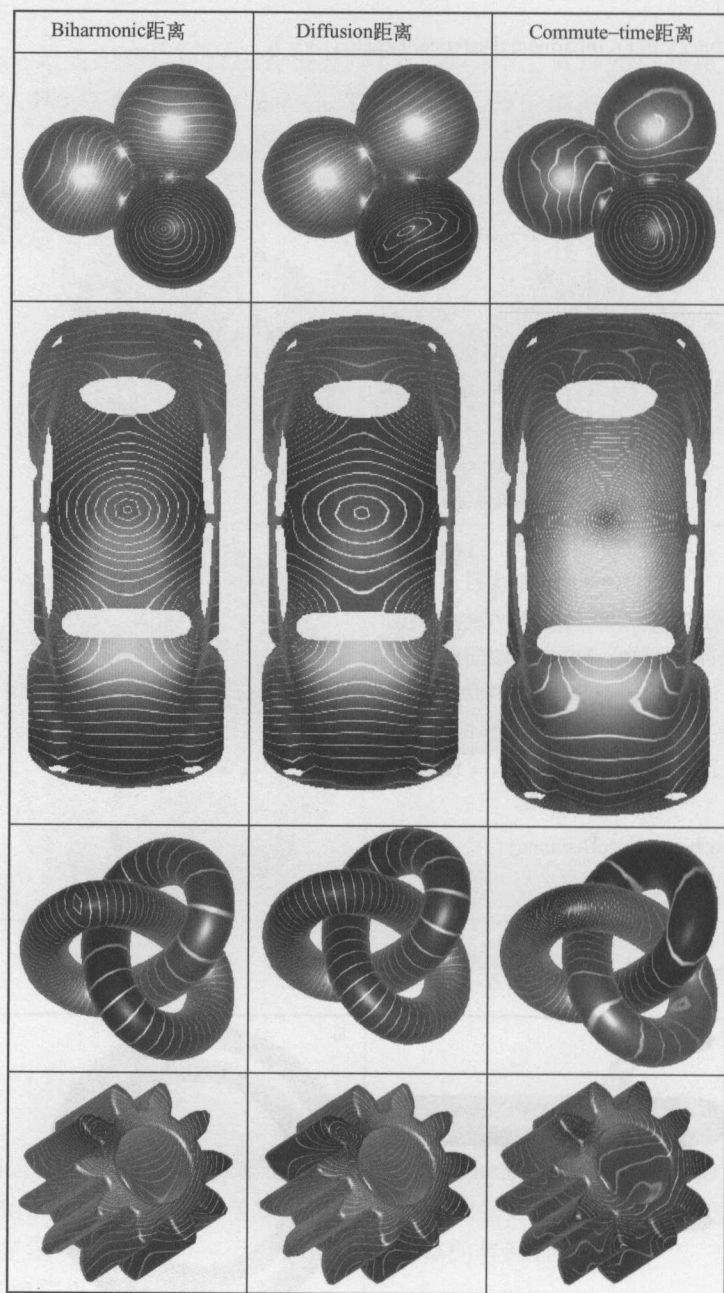


图 9-7 三种最短距离等值线对比

如图 9-8 所示是在同一球和有噪声的球两个三维模型上三种最短距离等值线的展示，从中可以看出 Biharmonic 在球形上的等值线分布是最好的，几乎是个正圆，随着距离的变远，等值线的圆形保持不变。Commute - Time 的等值线就有很大扭曲，和准确值相差较远。即使三维模型上有噪声，也不影响最短距离的计算。

如图 9-9 所示，在有洞的情况下，Biharmonic 最短距离能够很好地穿越洞，而 Commute - time 最短距离就受到洞的干扰较多。

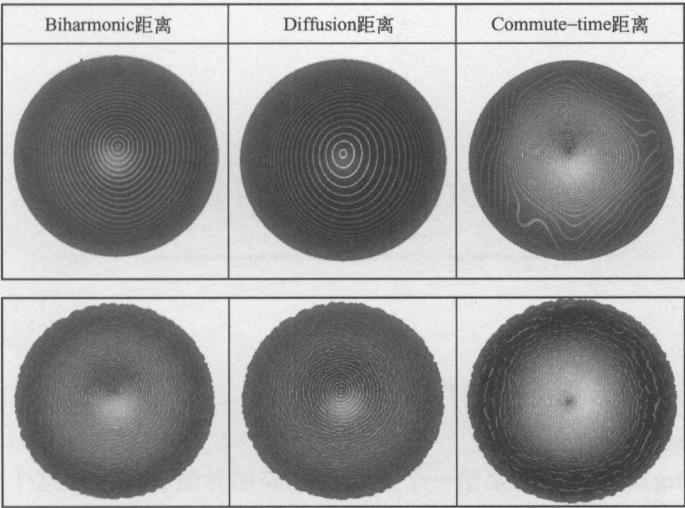


图 9-8 有噪声模型的最短距离等值线

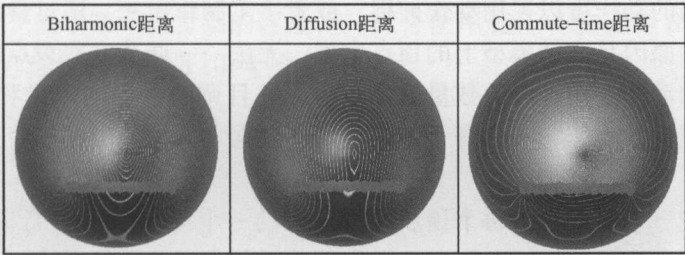


图 9-9 有洞模型的最短距离等值线

在原始模型和简化后的三维模型外观形状一样的情况下，假如原点形状也一样，那么最短距离等值线展示出类似的分布，但是 Biharmonic 最短距离的效果最好。例如，如图 9-10 所示，第一行是兔子的原始模型，顶点数量是 4968，第二行是简化后的模型，顶点数是 1000。

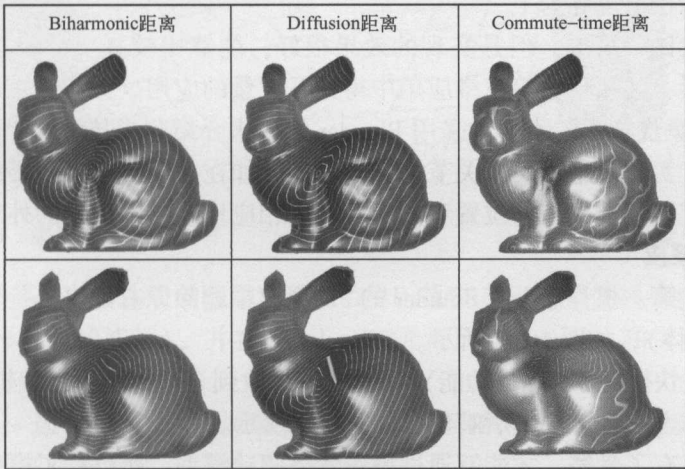


图 9-10 简化模型的最短距离等值线



10.1 两个关节的简单蒙皮

蒙皮 (Skinning) 技术指的是给一个三维模型添加骨骼，并且把这个骨骼和三维模型上的顶点绑定的过程。通过蒙皮技术可以使静止的三维模型随着骨骼动起来，从而变成动态的三维模型。例如，一个人物的三维模型，在和骨骼绑定之后，就可以具有跑步、跳跃、跳舞等各种动作。骨骼的动作可以采用动作捕捉，或者手工制作出来。通过骨骼驱动绑定的三维模型动作最重要的原因是三维模型上的顶点太多，无法一一调节这些顶点的位置，从而使三维模型具有不同的动作。而骨骼的数量就比较少，并且调节骨骼的位置和方向也比较直观，能很自然地得到想要的动作。在具有了骨骼的动作数据之后，蒙皮就是使三维模型动起来的核心和关键。

蒙皮技术中把三维模型上的每个顶点和骨骼中的一个或多个骨骼关联起来，一般来说，一个顶点可以关联 4 个骨骼，但是可以根据实际需要制定要关联的骨骼数量。每个顶点关联的骨骼都是和这个顶点比较临近的骨骼，因为这些顶点的位置受到临近骨骼顶点的影响而不会受到较远处骨骼的影响。

蒙皮技术中另一个重要的概念是**权重 (Weight)**。权重指的是顶点所关联的骨骼对此顶点影响的程度，权重越大，影响程度越大，权重越小，影响程度越小。这个顶点所有关联骨骼对此顶点影响的权重之和为 1。

蒙皮技术虽然比较简单，但是实现的效果很好，能够比较真实的表现人物、动物的动作，在游戏、电影、虚拟现实等三维应用中得到了大量的应用。大多数三维软件，如 Maya，3DSMax 等都支持蒙皮技术，本章中采用 Blender 软件来介绍蒙皮技术的界面和使用方法。

最简单的蒙皮实例是采用两个关节构成的骨骼，如在一个圆柱体三维模型中，添加两个关节，从而使改变这两个关节的位置和方向就可以相应地改变圆柱体的外观形状。

1. 自动权重蒙皮

第一步：建立第一根骨骼。在 Blender 的 3D 视窗中删除原有的 Cube（正方体），建立一个 Cylinder（圆柱体），如图 10-1 所示。

第二步：通过快捷键 S（缩放功能）将圆柱体缩放到合适大小，在编辑模式下用 Ctrl + R 横向切割圆柱，滑动滚轮可增加切割环，如图 10-2 所示。

第三步：只有有了骨骼，才能够通过操作骨骼驱动模型，所以有了模型后首先要先建立骨架。把鼠标移动到 3D 视窗上，然后按 Shift + A，选择 Armature -> Single Bone，如图 10-3 所示。

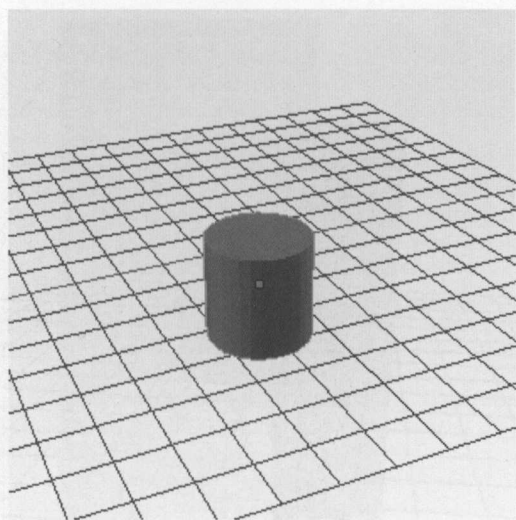


图 10-1 建立一个圆柱体

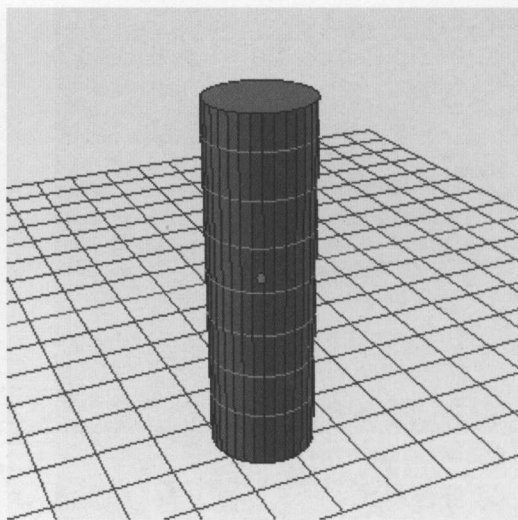


图 10-2 调整后的圆柱体

第四步：这样完成了第一根骨架的建立，并配合使用旋转（快捷键 R）、移动（快捷键 G）将骨架安放到合理位置，如图 10-4 所示。

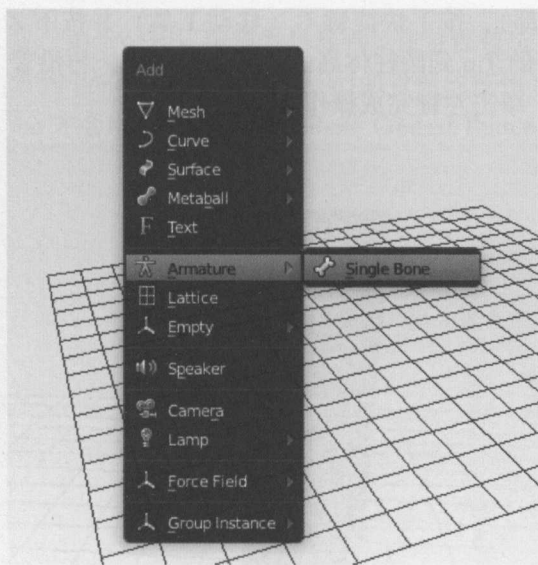


图 10-3 添加一个骨架

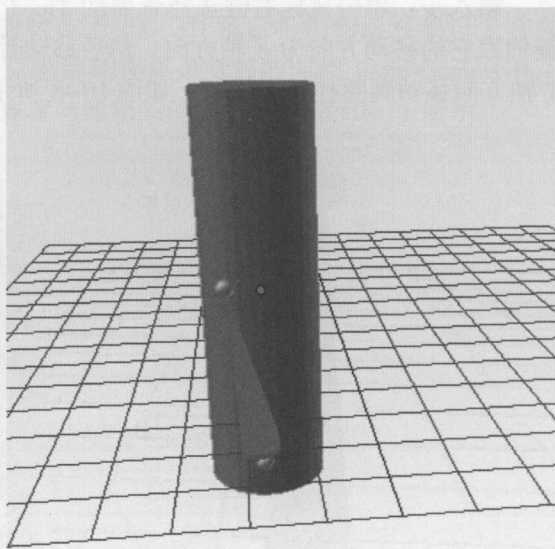


图 10-4 骨架安放

第五步：确保骨架在被选中的状态，然后在 3D 视窗右侧的属性视窗中单击 Object Context Button（像人形骨架的按钮），并在 Display 选项卡下勾选 Names 和 X-Ray，如图 10-5 所示。

第六步：Names 选项确保在 3D 视窗中显示骨架的名称，而 X-Ray 选项确保在 Object Mode 下也能透过模型看到骨架，如图 10-6 所示。

第七步：编辑骨骼，把鼠标移动到 3D 视窗上，按下 Tab 键从 Object Mode 转化到 Edit Mode，仔细观察建立的骨骼，每根骨骼由 3 部分组成，即 Root、Body、Tip，如图 10-7 所示。

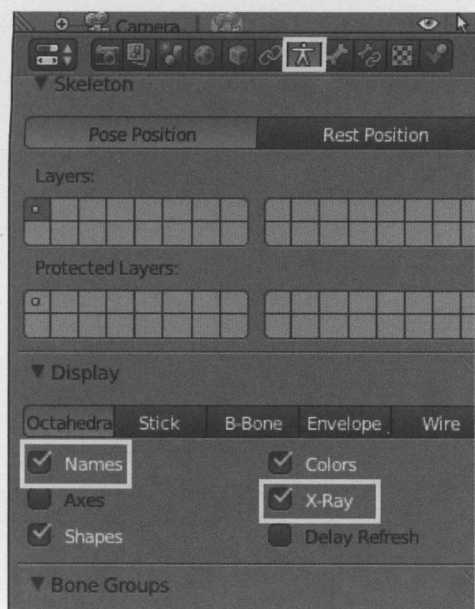


图 10-5 骨骼属性菜单

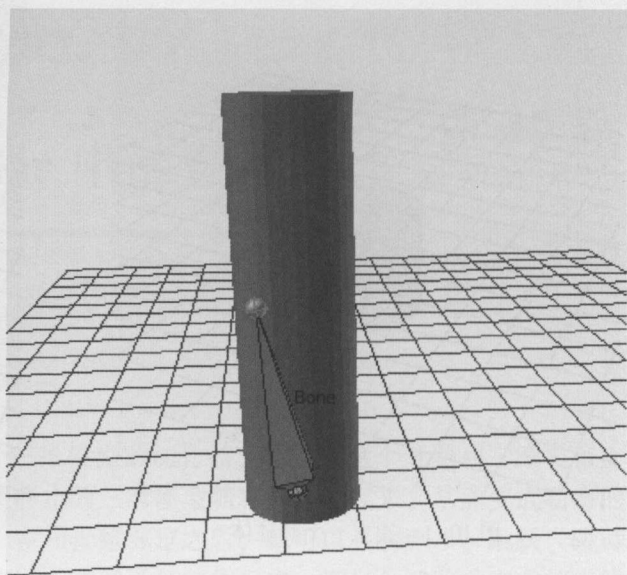


图 10-6 骨骼名称及透视

第八步：单击鼠标右键选择骨骼的 Tip（顶部），按下快捷键 G（移动工具）并按下 Z 键约束它的移动方向为 Z 轴方向，然后移动骨骼的 Tip 到圆柱体高的一半位置左右，再沿着 Y 轴方向移动到圆柱体边缘处，如图 10-8 所示，单击左键完成操作。

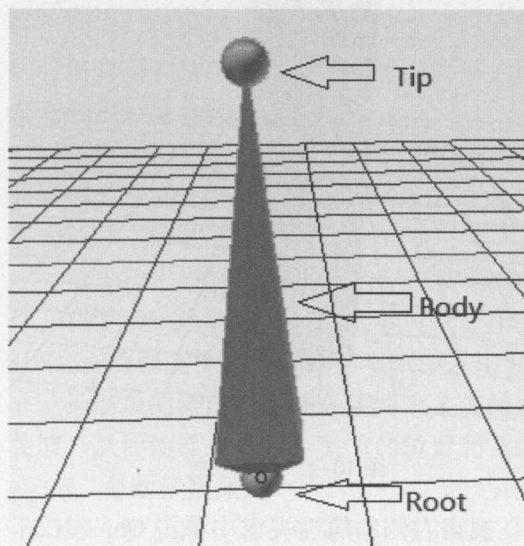


图 10-7 骨骼组成部分

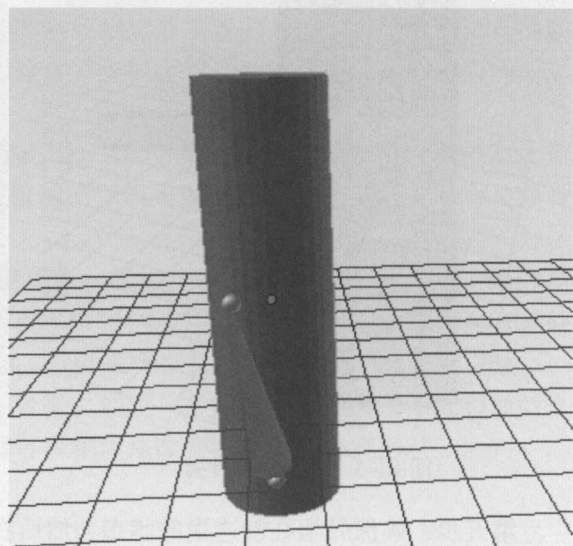


图 10-8 骨骼位置调整

第九步：生长骨骼。骨骼生长可以有两种方式，一是通过 3D 视窗左侧的菜单下的 Armature Tools 工具的 Extrude 命令，如图 10-9 所示。

第十步：第二种生长骨骼方法是通过快捷键 E 进行骨骼生长。按下 E 键进行骨骼生长并滑动鼠标到圆柱体的左上角处，单击左键完成第二根骨骼的建立，如图 10-10 所示。

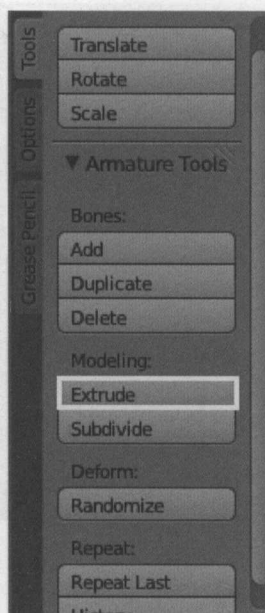


图 10-9 生长骨骼方法一

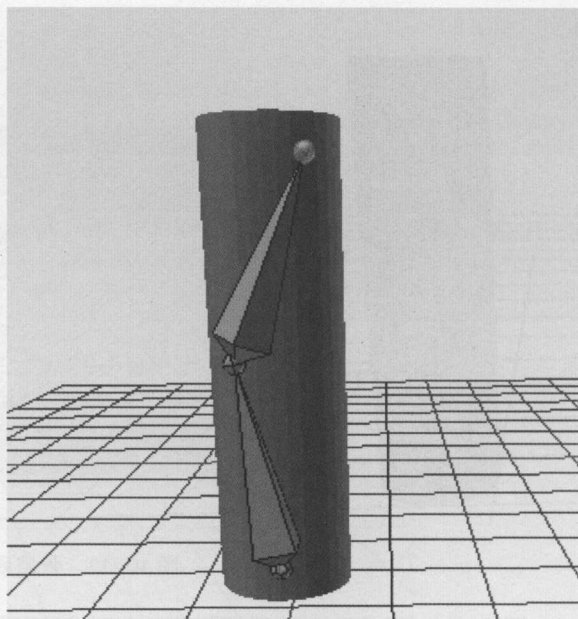


图 10-10 生长骨骼方法二

第十一步：编辑骨骼属性。在骨骼被选中的状态下，在 3D 视窗右侧的属性窗口上方会看到 Bone Context Button（形似骨骼的按钮），只有活动物体是骨骼的时候 Bone Context Button 才会出现，左键单击 Bone Context Button，查看与此相关的属性，如图 10-11 所示。

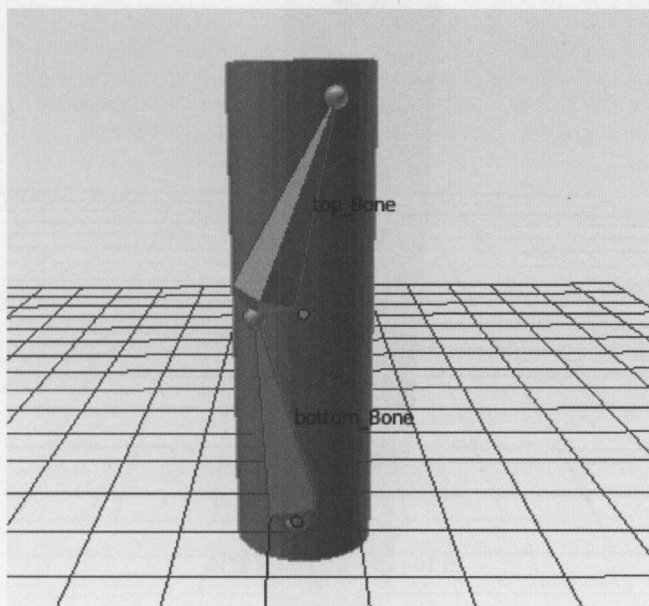


图 10-11 骨骼属性编辑 (1)

第十二步：在 3D 视窗上右键单击名为 Bone 的骨骼，在右侧属性面板的顶部修改骨骼名称为 bottom_Bone，然后选择 Bone.001 骨骼并重新命名为 top_Bone，如图 10-12 所示。

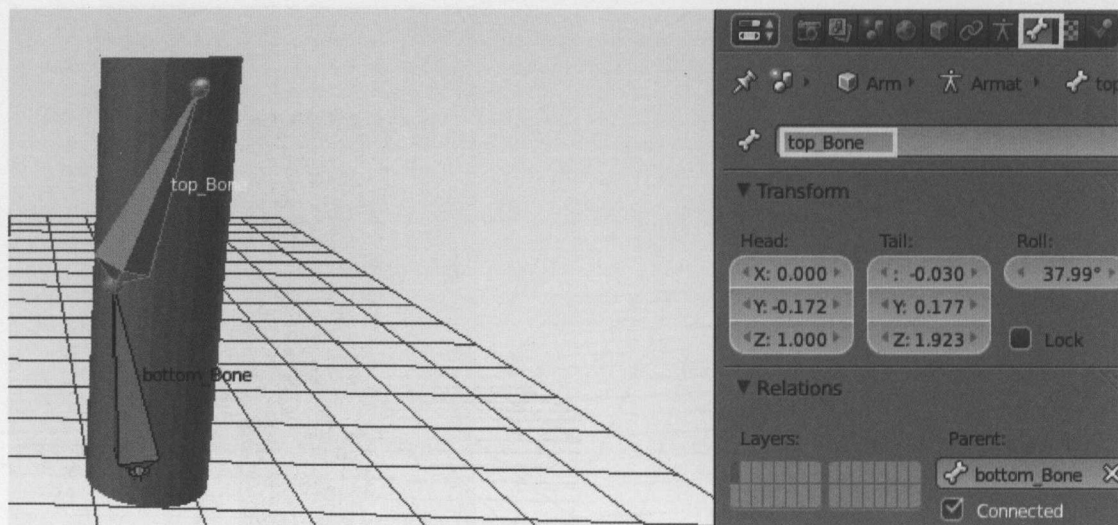


图 10-12 骨骼属性编辑 (2)

第十三步：在骨骼和模型都放置好之后需要建立骨架和模型之间的关系。在 Object Mode 下右键选择模型，然后按住 Shift 右键继续选择骨架，这样就把模型与骨架全部选中，呈红色显示，如图 10-13 所示。

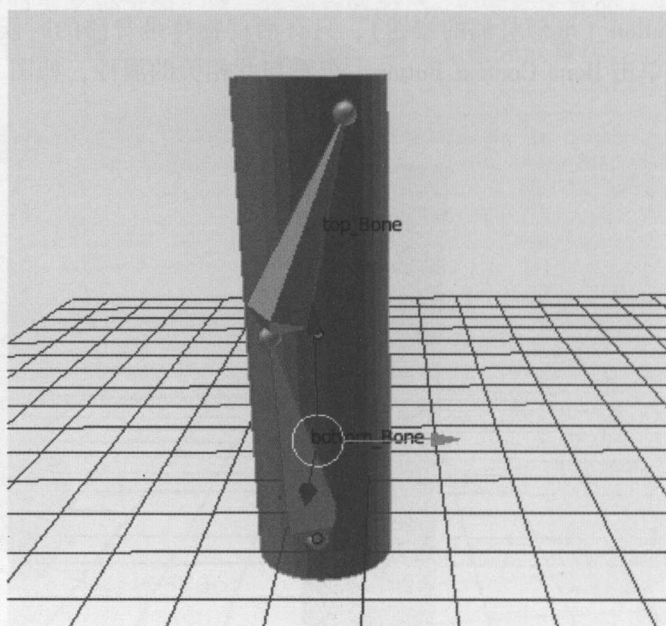


图 10-13 选中所有物体

第十四步：然后按 Ctrl + P 显示 Set Parent To 菜单，选择 With Automatic Weights 选项，如图 10-14 所示。

这样就设定了骨骼与模型的父子关系，并确定了权重的分配，因为采用了自动权重，因此骨骼对它周围顶点的影响程度是自动分配的，这就是说 Blender 会根据骨骼的邻近程度自

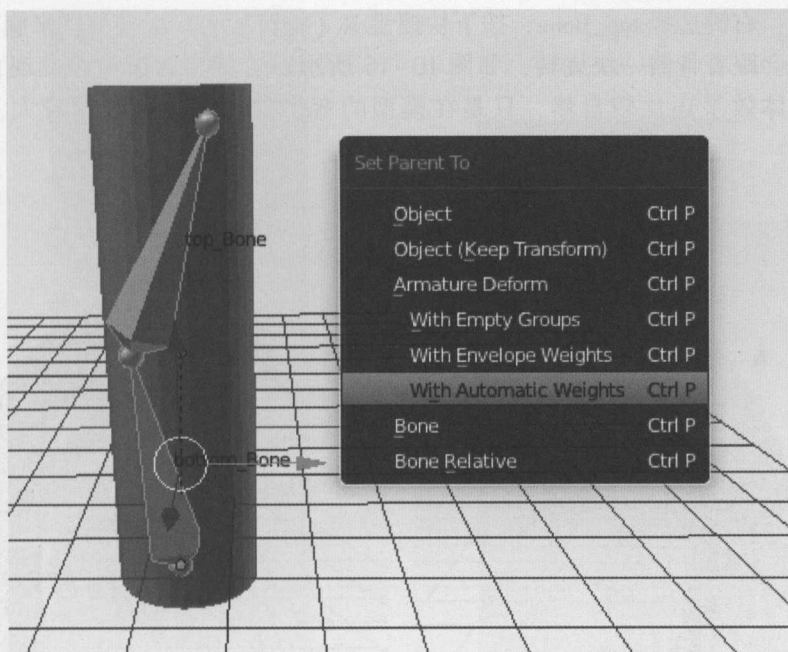


图 10-14 骨骼绑定

动确定它对周围顶点的影响程度。

第十五步：调整姿态。右键选择骨骼然后按 Ctrl + Tab 键进入 Pose Mode，或者从模式菜单中选择 Pose Mode，注意被选中的骨骼呈现浅蓝色，如图 10-15 所示。

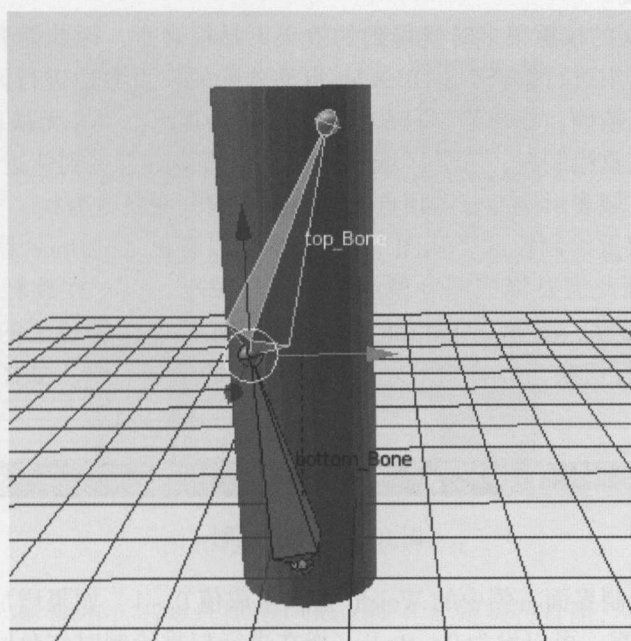


图 10-15 选中骨骼

第十六步：右键选择 `top_Bone`，按下快捷键 R（旋转功能）滑动鼠标来旋转骨骼，模型的上半部分也会跟着骨骼一块旋转，如图 10-16 所示通过旋转骨骼可以发现自动分配的权重的效果，整体效果还比较自然，只是在模型的弯折处以下部分变形较大，如图 10-17 所示。

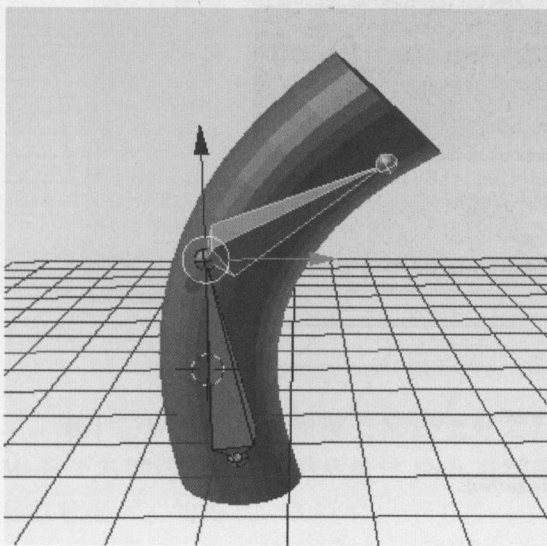


图 10-16 姿态调整 1

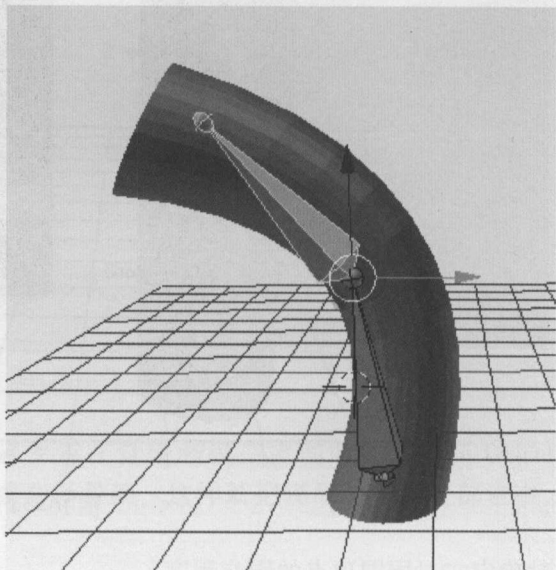


图 10-17 姿态调整 2

2. 绘制权重蒙皮

Blender 自动分配的权重很多时候得到的效果不是很满意，因此需要绘制权重。权重绘制就是对模型的顶点组进行编辑，对 Blender 自动分配的权重重新进行调整，从而使顶点受到骨骼影响的效果更精确。权重绘制只是作用在模型的顶点上，在无顶点处刷权重不会起到任何效果。在进行权重绘制时，红色区域表示该部分的顶点组受到邻近骨骼的影响程度大，即权重大。而蓝色区域表示该部分的顶点组并未被影响，即权重为 0。

第一步：进入权重绘制模式。从图 11-17 中可以看出 `top_Bone` 骨骼影响到了模型下半部分的顶点组，因此需要将这部分顶点组的权重减小，以求在旋转 `top_Bone` 骨骼时模型的下半部分不会变形。按下 `Ctrl + Tab` 键或使用 3D 视窗标题栏上的模式下拉菜单选择 `Weight Paint` 菜单进行切换，即可以显示出权重绘制的笔刷工具，如图 10-18 和图 10-19 所示。



图 10-18 权重绘制

第二步：权重绘制界面右侧中的 `Weight` 指权重取值 $0 \sim 1$ ，如果增加权重，可以在此处设为 1，如果减小权重，可以设为 0；`Radius` 指在进行权重绘制时红色圆圈的半径，这个可以根据需要自行设定；`Strength` 指强度，即刷权重时的力度，强度越大，刷一下越明显。在这里，把 `Weight` 改为 0，`Radius` 改为 20px、`Strength` 改为 0.5。如图 10-20 所示。



图 10-19 权重绘制工具栏



图 10-20 权重绘制界面

第三步：选择 top_Bone，然后开始用笔刷涂抹或是单击模型的下半部分使其颜色变成蓝色，最终刷权重前后对比如图 10-21 所示。

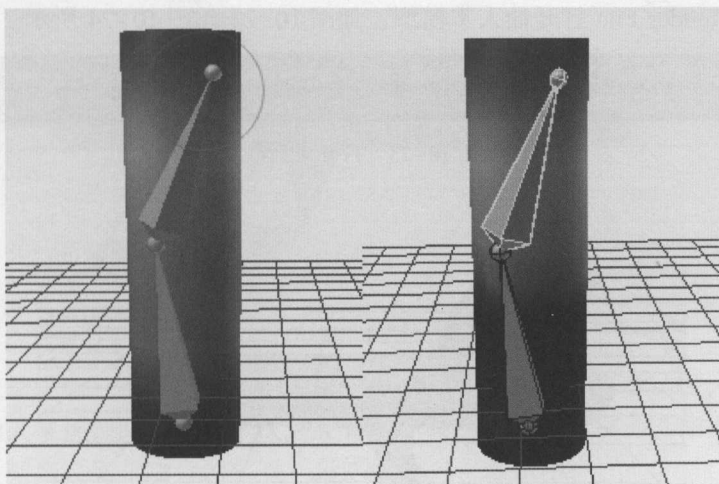


图 10-21 权重绘制前后对比

第四步：进入 Pose Mode，右键选择 top_Bone 按下 R 键旋转骨骼，与之前未调整权重时的对比如图 10-22 所示。

通过图 10-22 中可以发现，对模型下半部分顶点组权重的再绘制，就可以实现再次旋转 top_Bone 时模型下半部分不再受 top_Bone 影响。

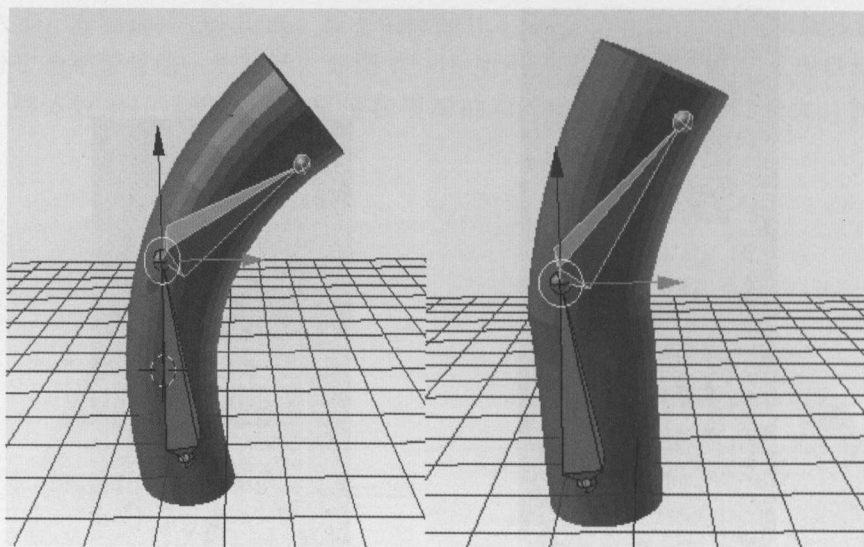


图 10-22 调整模型姿态对比



10.2 仙人掌蒙皮

上一节两个关节的圆柱蒙皮是最简单的蒙皮，这一小节在比较复杂的仙人掌三维模型上进行蒙皮技术。仙人掌三维模型的形状更复杂，因此需要的骨骼更多，但是基本步骤是一样的。第一步仍然是在原有模型的基础上建立骨架，然后编辑骨骼，生长骨骼，最后再进行权重绘制。

第一步：打开 Blender 软件，单击标题栏的 File -> import，找到模型文件所在的路径，然后单击 Open Blender File 打开仙人掌模型，如图 10-23 和图 10-24 所示。

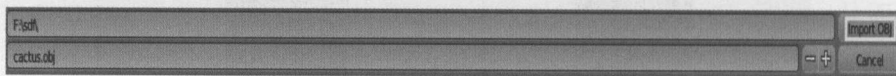


图 10-23 打开对话框

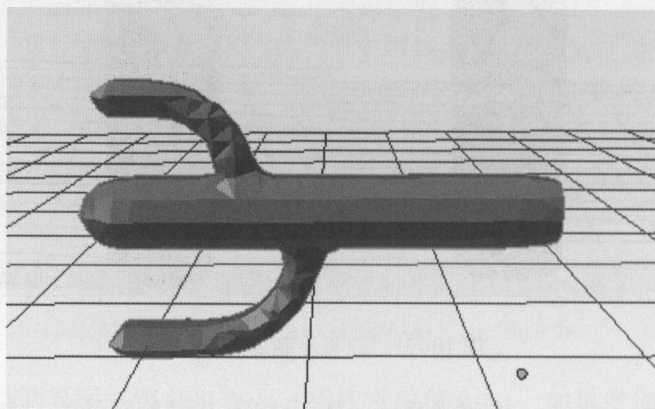


图 10-24 加载仙人掌三维模型

第二步：建立骨骼。按下快捷键 Z 使模型从 Solid 模式切换到 Wireframe 模式，在模型的右端单击左键，按 Shift + A 弹出 Add 菜单，如图 10-25 所示。

第三步：选择 Armature -> Single Bone，第一根骨骼就在此处建立，如图 10-26 所示。



图 10-25 添加骨骼菜单

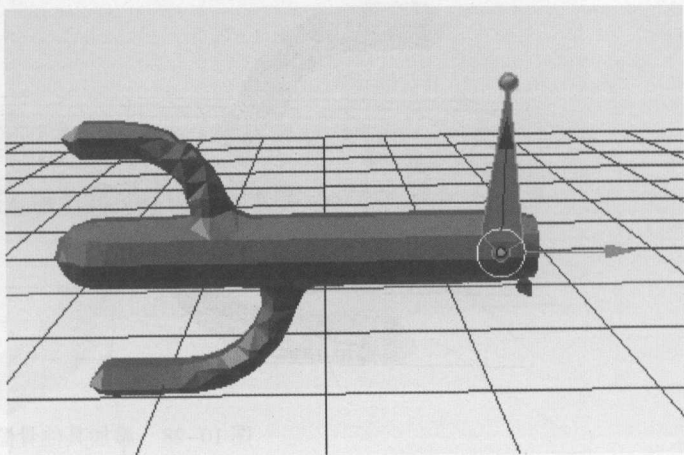


图 10-26 第一根骨骼

第四步：编辑骨骼。按下 Tab 键进入 Edit Mode 可以进行骨骼编辑，使用旋转功能（快捷键 R），按下 R 键再按 X、Y、Z 可约束旋转方向沿着各个坐标轴，调整方向使其沿着模型的方向，配合使用移动功能（快捷键 G），按下 G 再按 X、Y、Z 可约束移动方向沿着各个坐标轴，左键单击骨骼的 Tip 滑动鼠标可实现横向的缩放，最终使骨骼位置和形态如图 10-27 所示。

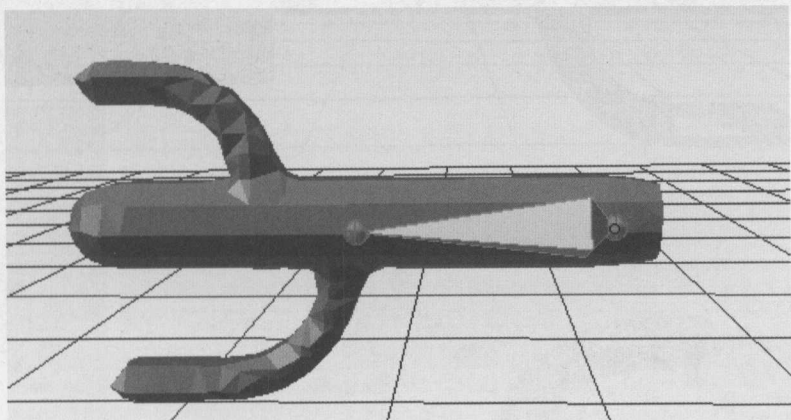


图 10-27 调节骨骼位置和方向

第五步：生长骨骼。在 Edit Mode 下按 E 键进行骨骼生长并滑动鼠标，单击左键完成骨骼的建立，从仙人掌颈到两个枝丫依次建立骨骼，并配合旋转、移动操作不断调整骨骼的位置，最终达到骨架与模型相吻合的效果，最终如图 10-28 所示。

第六步：这一步进行骨骼绑定，建立骨架和模型之间的关系，在 Object Mode 下右键选择模型，然后按住 Shift 右键继续选择骨架，这样就把模型与骨架全部选中，然后按 Ctrl + P 显示 Set Parent To 菜单，选择 With Automatic Weights 选项，如图 10-29 所示。

第七步：进行姿态调整。右键选择骨骼然后按 Ctrl + Tab 键进入 Pose Mode，或者从模式菜单中选择 Pose Mode，被选中的骨骼呈现浅蓝色，如图 10-30 (a) 所示；选择上面那根枝丫的最底部的那根关节左右摇摆一下，发现枝丫两侧模型变形较大，如图 10-30 (b) 所示。

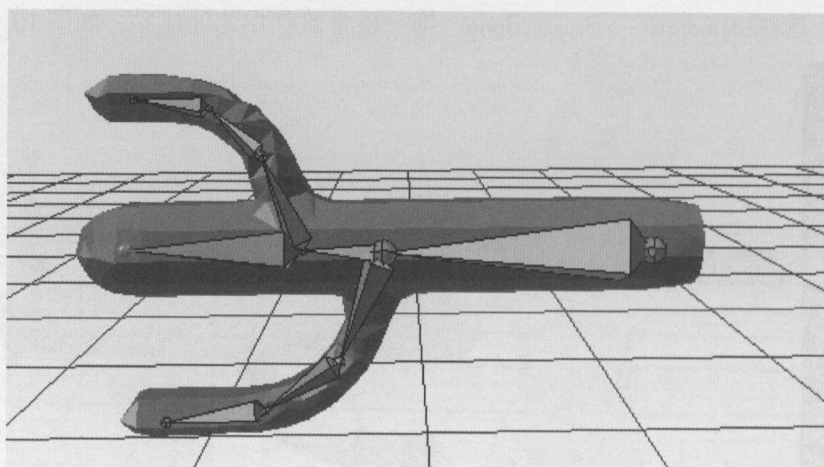


图 10-28 添加其他骨骼

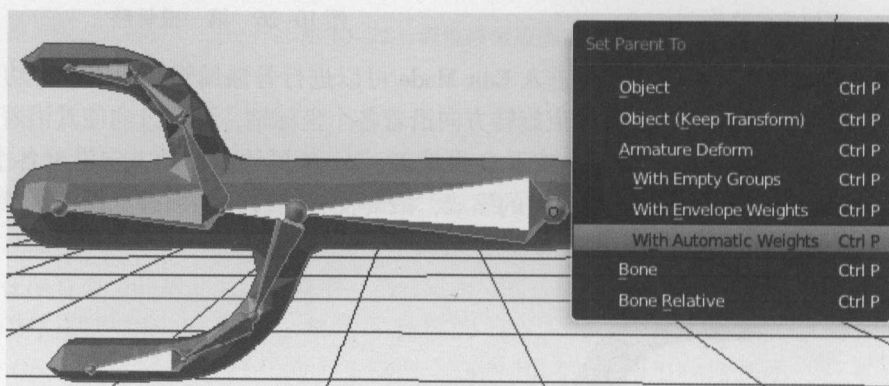


图 10-29 自动权重绑定

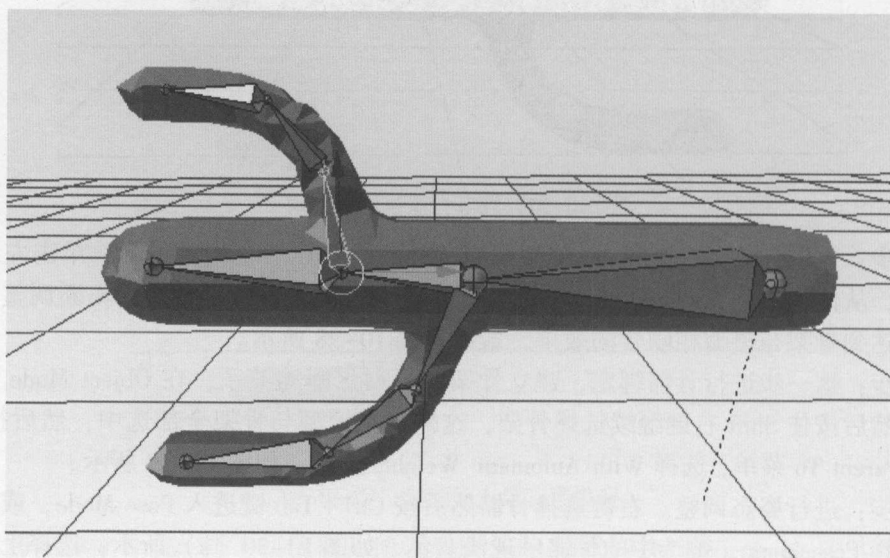


图 10-30 (a) 选中一个骨骼

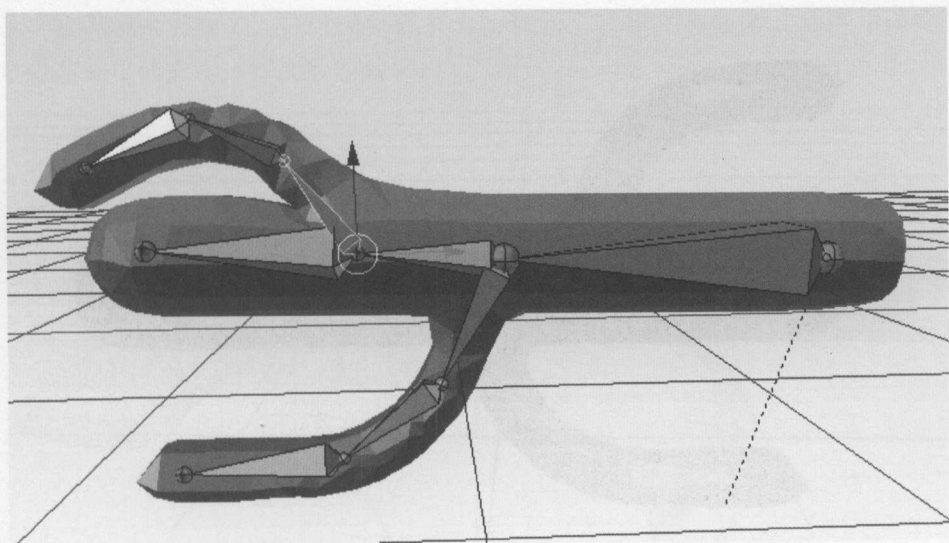


图 10-30 (b) 调整选中骨骼位置

第八步：进行绘制权重。按下 **Ctrl + Tab** 键或是使用 3D 视窗标题栏上的模式下拉菜单选择 **Weight Paint** 进行切换，使用笔刷工具进行权重绘制。在这里需要解决的问题是骨骼的旋转影响到了不应该影响的顶点组，所以要做的是减小顶点组所受到的权重，把 **Weight** 设为 0、**Radius** 设为 20px、**Strength** 设为 0.3，如图 10-31 所示。

第九步：然后右键选择上枝丫最底部的那根骨骼，把红色圆形笔刷移动到此处涂抹或单击不应该受影响的顶点组，刷完权重前后对比如图 10-32 和图 10-33 所示。



图 10-31 绘制权重菜单

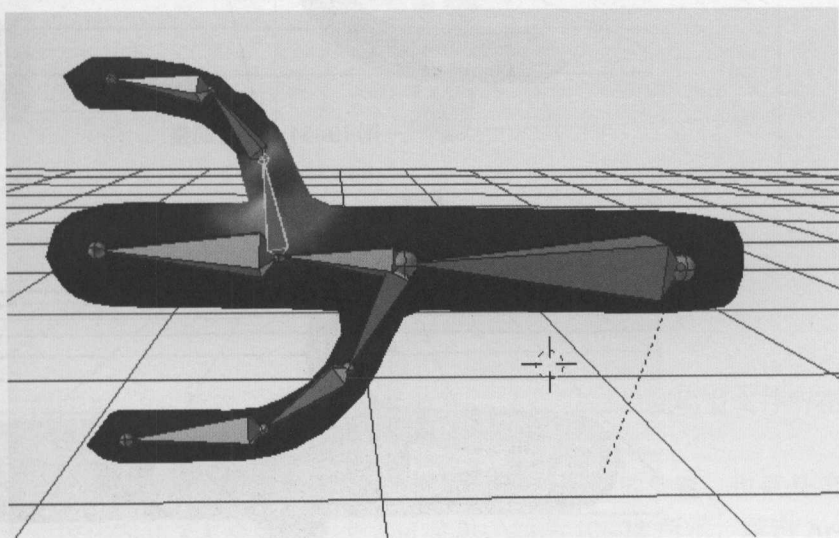


图 10-32 权重绘制前

第十步：通过上图对比我们发现，上枝丫最底部那根骨骼周围的权重颜色由黄绿色变为蓝色，即权重由小变大。右键选择这根骨骼按下 **R** 键向左、向右旋转，效果图如图 10-34 和图 10-35 所示。现在骨骼左右两侧模型没有了较为明显的变形。

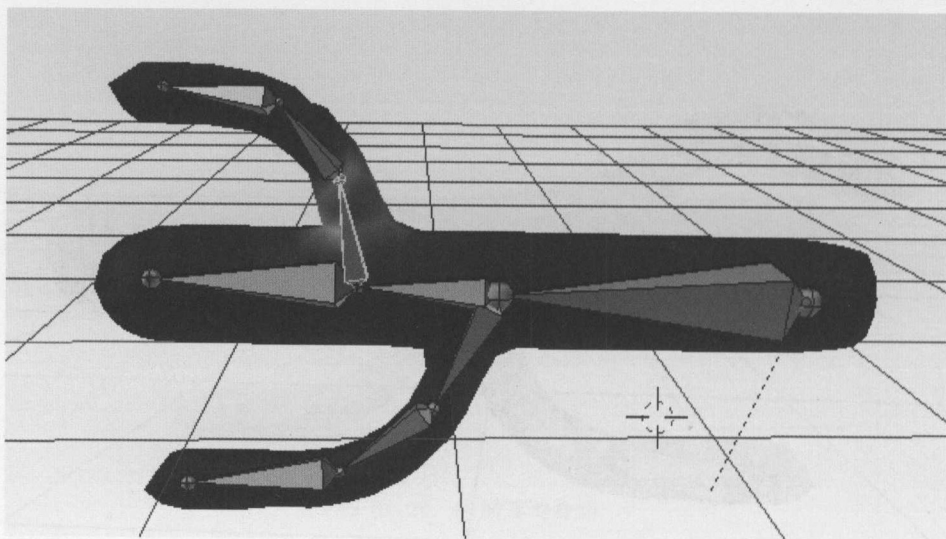


图 10-33 权重绘制后

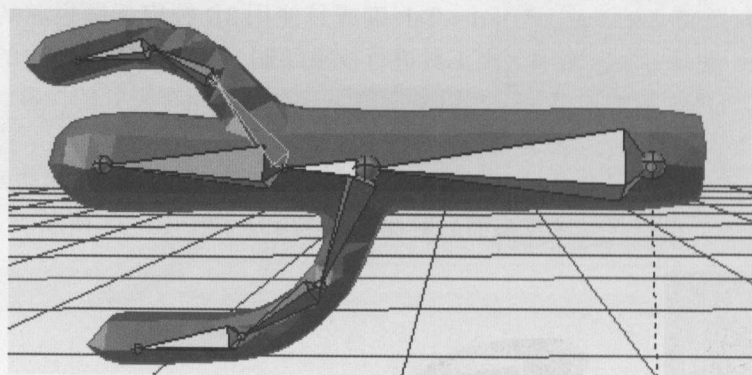


图 10-34 向左调整

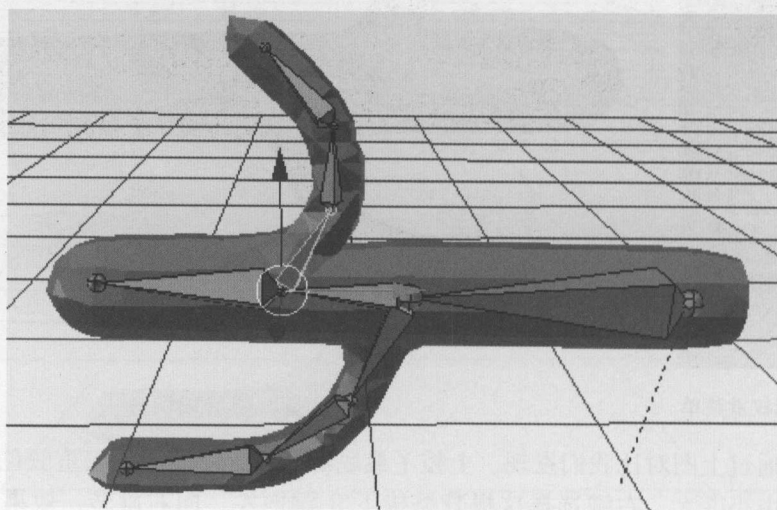


图 10-35 向右调整

第十一步：对于下枝丫，可以用同样的方法调整，这里不再重复。如图 10-36 所示是各种通过骨骼蒙皮仙人掌进行变形的效果。

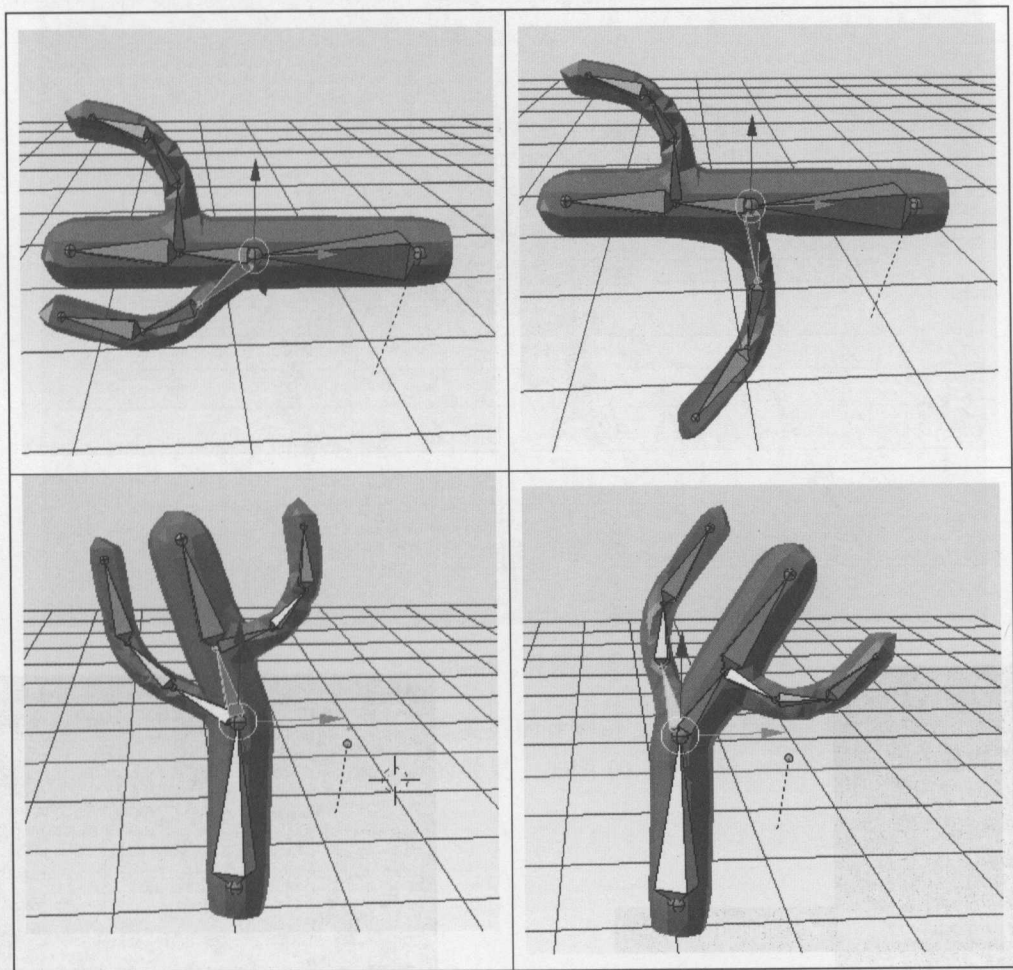


图 10-36 姿态调整



10.3 马匹蒙皮

第一步：导入模型。打开 Blender 软件，单击标题栏的 File -> Open，找到模型文件所在的路径，然后单击 Open Blender File 打开马匹模型，如图 10-37 所示。

第二步：建立骨骼。在模型的某一部位单击一下鼠标左键，给定准星的位置，也就是第一根骨架就在准星处建立。按住 Shift + A 会弹出一个 Add 菜单，如图 10-38 所示，单击 Armature -> Single Bone，完成第一根骨架的建立。

第三步：确保骨架在被选中的状态，然后在 3D 视窗右侧的属性视窗中单击 Object Context Button（像人形骨架的按钮），并在 Display 选项卡下勾选 X-Ray，如图 10-39 所示。并配合使用旋转（快捷键 R）、移动（快捷键 G）将骨架安放到合理位置。

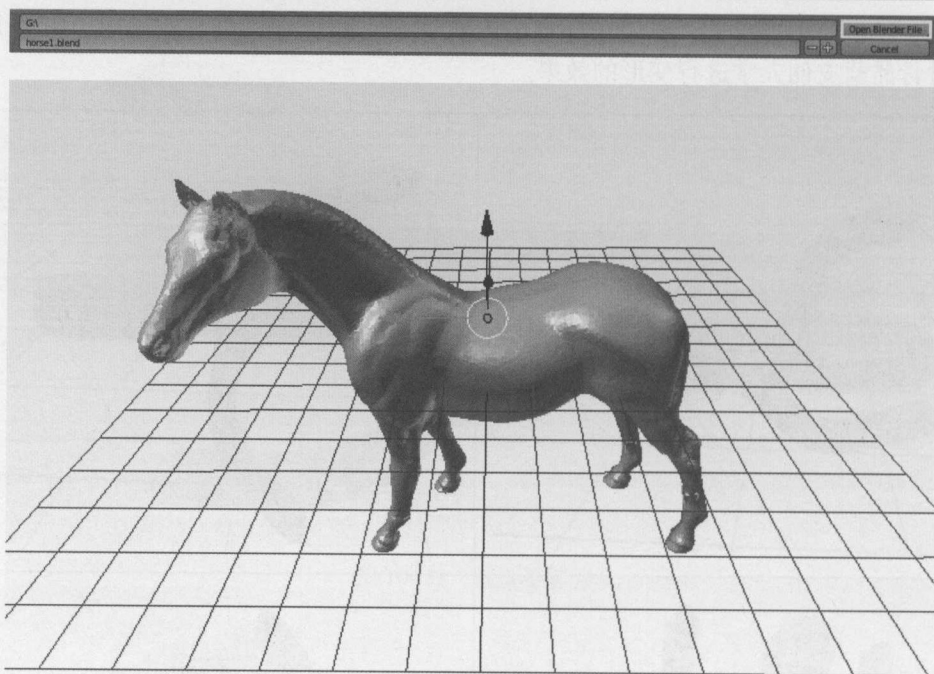


图 10-37 马匹蒙皮

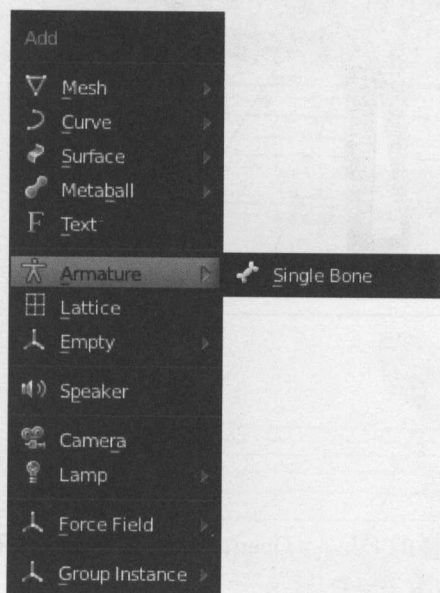


图 10-38 添加骨骼菜单

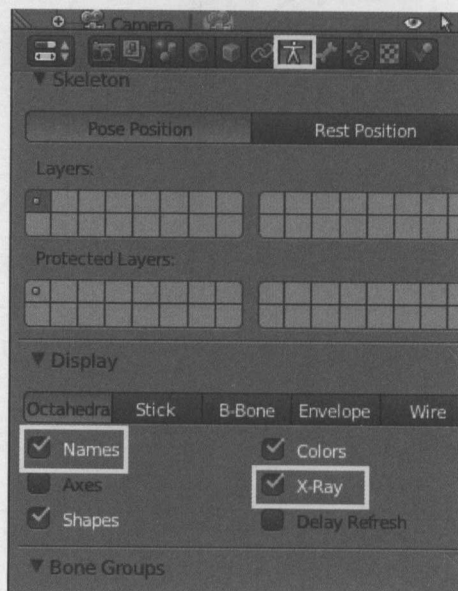


图 10-39 骨骼属性菜单

第四步：按“Tab”键从 Object Mode 切换到 Edit Mode，在该模式下可以进行骨骼的编辑操作。选中刚刚建立的骨架，按 E 键进行骨骼的生长，再单击左键完成一次骨骼生长，这样骨骼就从一节生长到两节，如图 10-40 所示。

第五步：按照第四步同样的方法，按 E 键继续骨骼的生长，从马匹的背部到头部再到

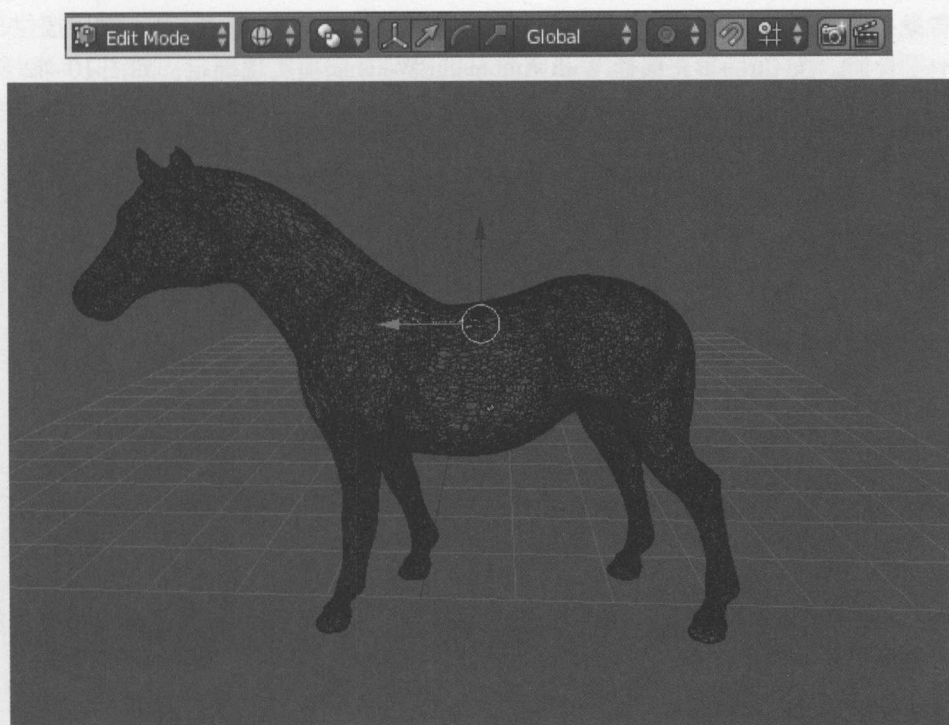


图 10-40 骨骼生长

腿部，依次建立骨骼，形成具有父子层次关系的骨骼链，并配合旋转、移动操作不断调整骨骼的位置，最终达到骨架与模型相吻合的效果，如图 10-41 所示。

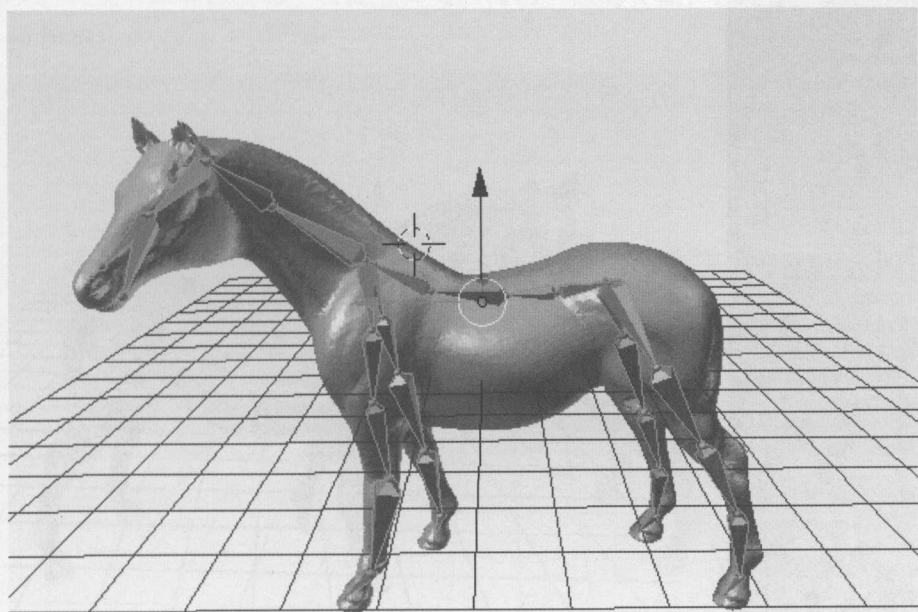


图 10-41 生成全部骨骼

第六步：使用自动权重绑定骨架。使骨骼在 Pose Mode 下先选定模型，然后按住 Shift 选定任意一根骨骼，按 Ctrl + P，选择 With Automatic Weights 项实现绑定，如图 10-42 所示。

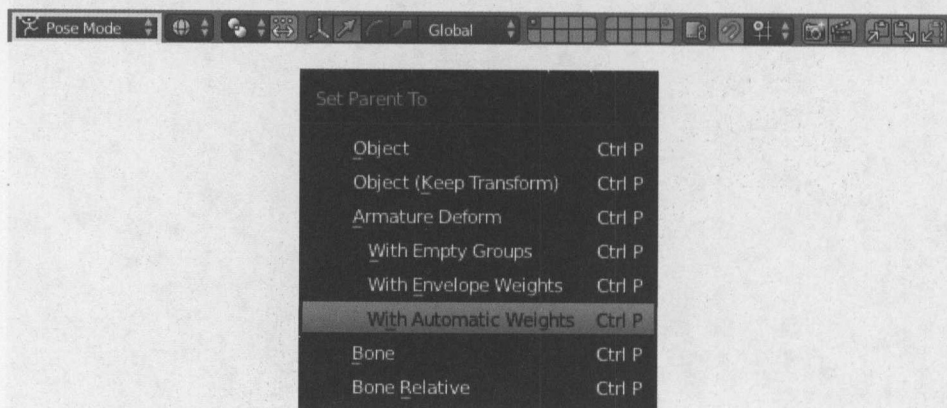


图 10-42 自动绑定

第七步：在 Display 中选择 X-Ray，从而可以透过模型看到骨架以便操作，此时就可以通过移动或旋转骨架实现三维模型的变形和动作，如图 10-43 所示和图 10-44 所示。

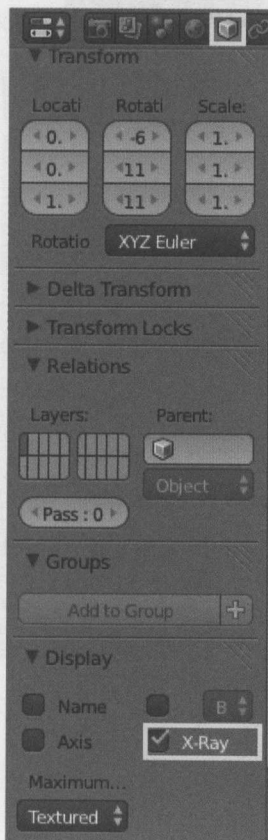


图 10-43 骨骼属性设置菜单

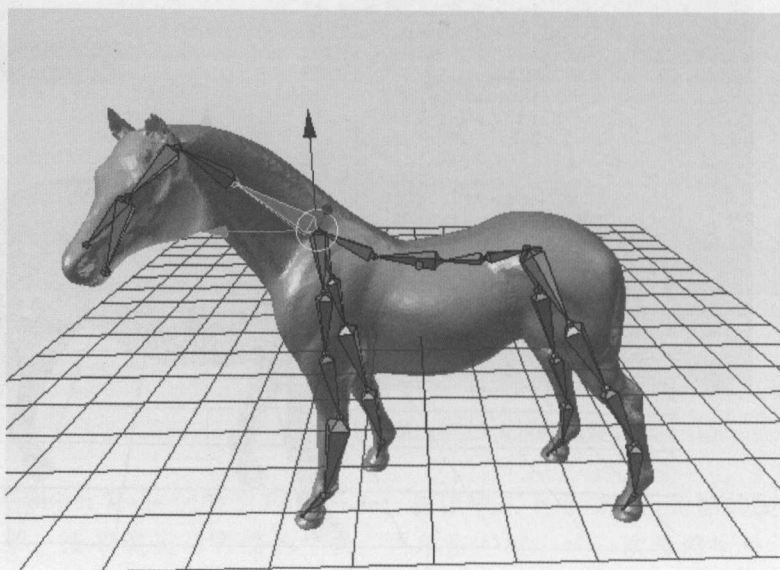


图 10-44 骨骼绑定结果

第八步：调整马头姿态。右键选择骨骼然后按 Ctrl + Tab 键进入 Pose Mode，或者从模式菜单中选择 Pose Mode，按下快捷键 Z 使物体呈现网格，右键选择马匹颈部的一根骨骼，按下 R 键旋转骨骼，如图 10-45 所示。在旋转骨骼的时候发现马匹颈部的旋转并没有引起与颈部紧邻的背部的移动，通常情况下马在抬起头或是低头时靠前的背部也会有变化的，这是采用自动权重带来的权重和骨骼不匹配所带来的问题，需要用手工权重来调节。

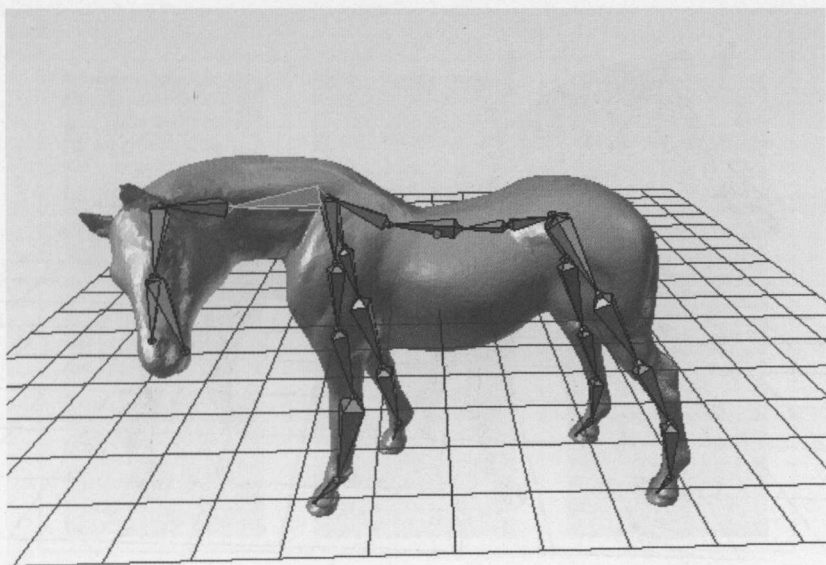


图 10-45 调整马头

第九步：调整马匹嘴巴姿态。右键选择马匹嘴巴下面的骨骼，按下 R 键做适当的旋转，如图 10-46 所示。

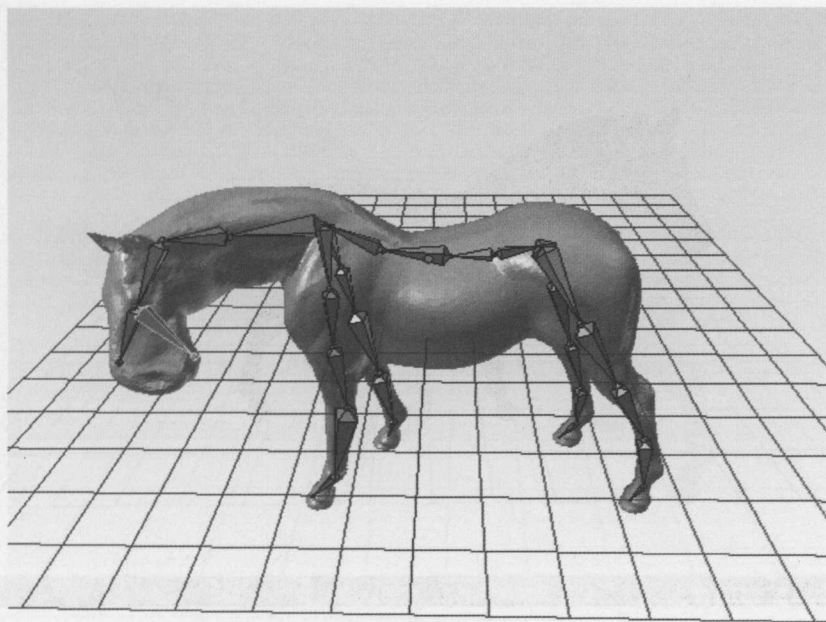


图 10-46 调整马嘴巴

第十步：在完成骨架的绑定后，可以通过操作骨架来实现模型的运动，但是有时模型某些部位的动态效果与实际情况不相符，这就需要重新修改骨骼的权重，即调整骨骼的影响范围，以此达到合理的动态效果。设定骨骼在 Pose Mode 下，如图 10-47 所示。

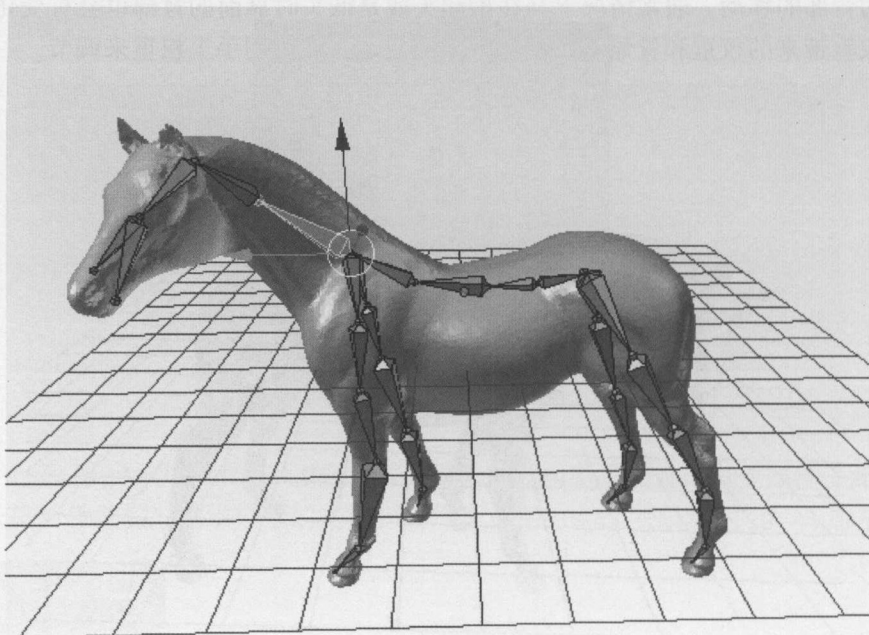


图 10-47 在 Pose 模式下

第十一步：右键选择模型，然后在模式菜单下选择 Weight Paint，进入权重绘制界面，如图 10-48 所示。

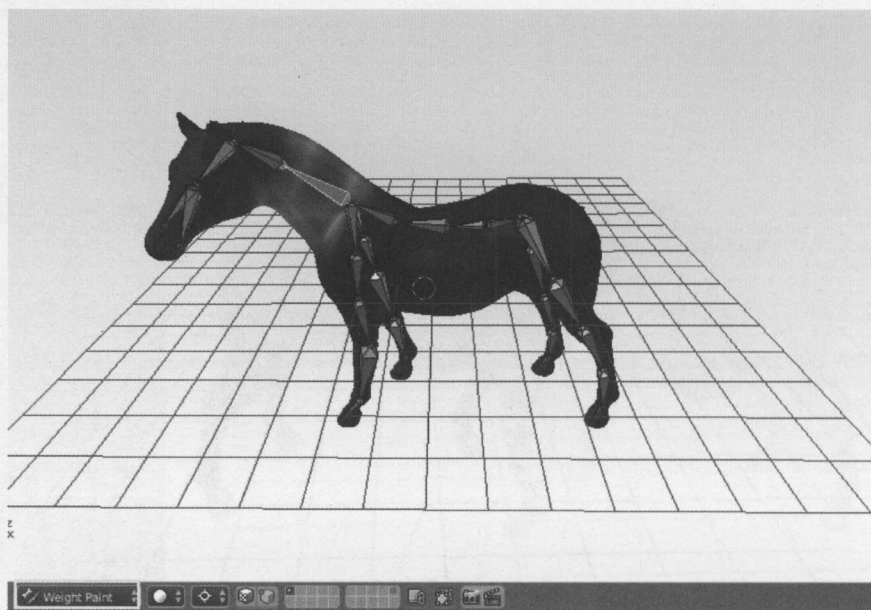


图 10-48 权重绘制

第十二步：需要在颈部的骨骼上增加些权重，选择颈部骨骼，将红色圆圈的画笔移动到要涂抹的位置，鼠标滑动或单击可以实现权重绘制，红色代表权重最大，蓝色代表权重最小，中间色表示权重渐变。在这里分三次刷权重，将 Weight 设为 1、Radius 设为 20px，第一次将 Strength 设为 0.3，第二次将 Strength 设为 0.2，第三次将 Strength 设为 0.1，以突出权重的逐级递减，如图 10-49 所示。

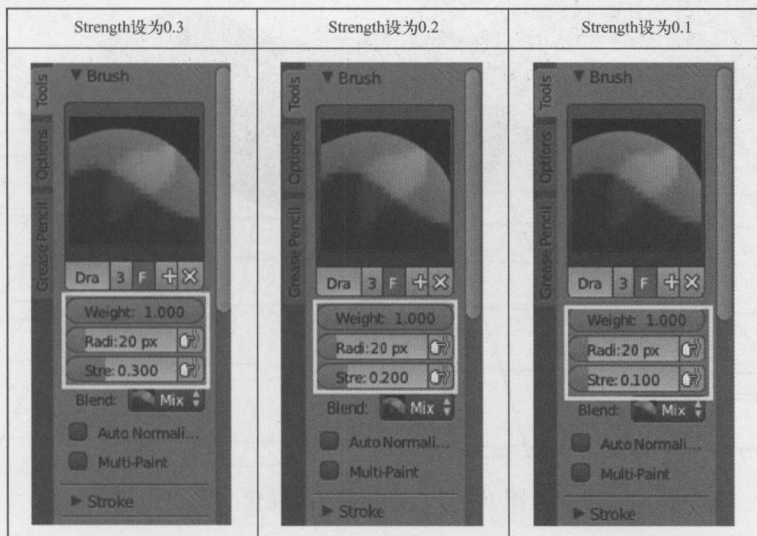


图 10-49 权重绘制菜单参数设定

第十三步：对比刷权重后的变化，如图 10-50 和图 10-51 所示。

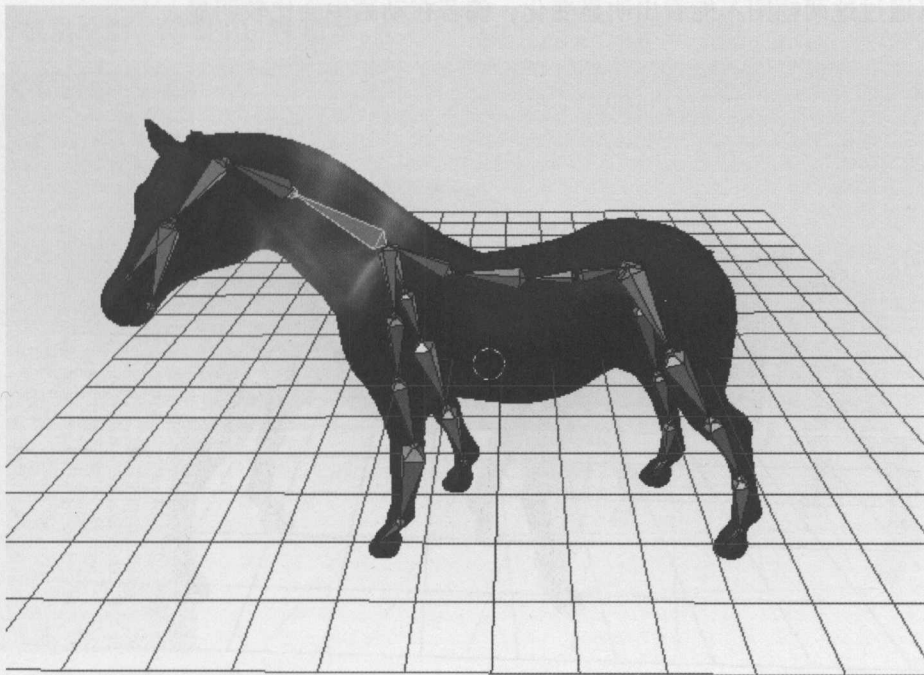


图 10-50 刷权重对比变化

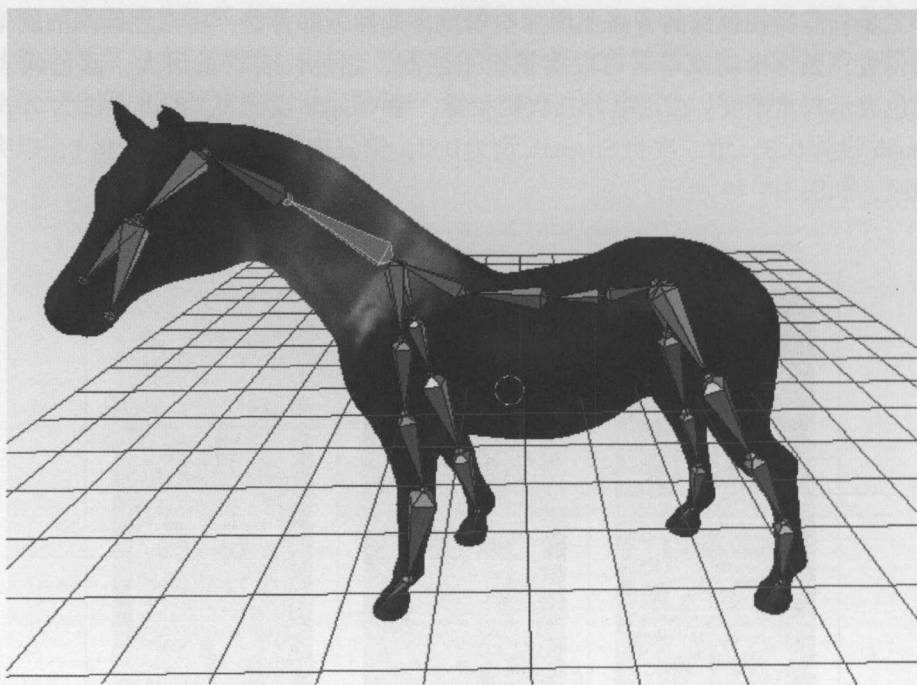


图 10-51 刷权重对比变化

第十四步：通过对比，发现颈部后面，马匹背部的权重颜色由深蓝色变为黄绿色，即权重由小变大，说明马匹颈部骨骼会影响马匹背部。进入 Pose Mode 右键选择颈部骨骼，按下 R 键旋转骨骼，与之前未调整权重时的对比如图 10-52 所示。由于这只是一匹马的扭脖子动作，所以通过这两幅图不能看出明显变化，倘若在动画中会比较明显。

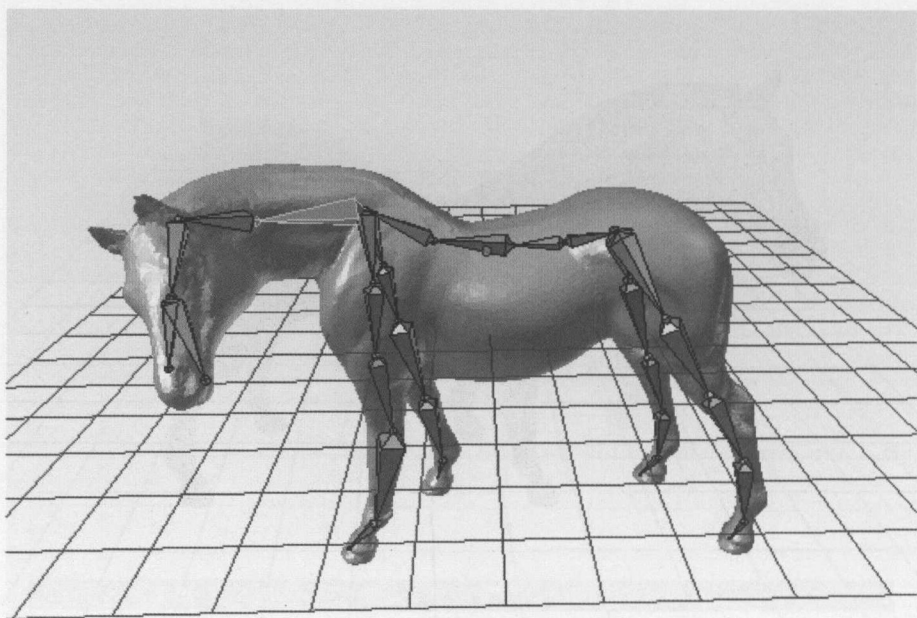


图 10-52 手工权重姿态调整对比

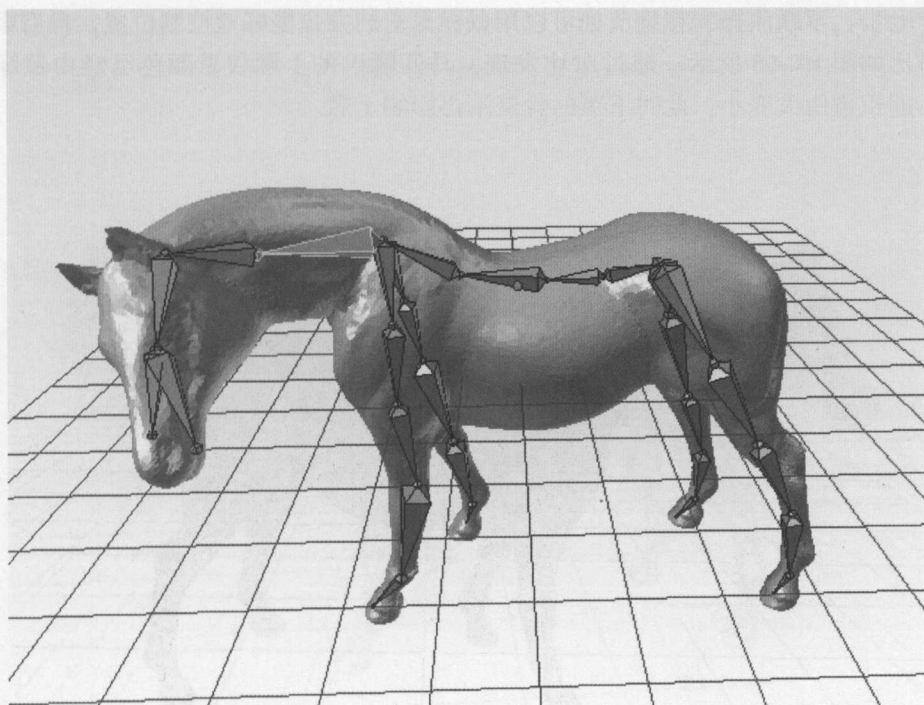


图 10-52 手工权重姿态调整对比 (续)

第十五步：对下颚骨骼的操作影响了上颚模型顶点组，需要减小上颚顶点组的权重，在这里我们把 Weight 取值设为 0、Radius 为 20px、Strength 取值为 0.3，如图 10-53 所示。

第十六步：选择下颚骨骼，将红色圆圈的刷笔移动到要涂抹的位置，在该处滑动鼠标或单击进行权重绘制，如图 10-54 所示。



图 10-53 权重绘制参数设定

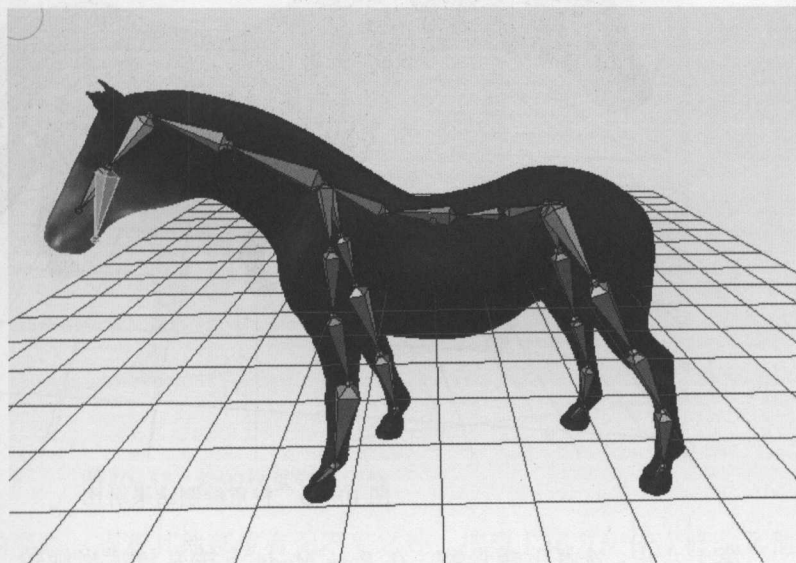


图 10-54 权重绘制

第十七步：不断涂抹，在细节处可以用鼠标单击的方式更好地控制位置，最后刷完权重的效果对比如图 10-55 所示。通过对比发现，马匹嘴巴的上颚权重颜色已经由黄绿色变为深蓝色，即权重由大变小，此时下颚的骨骼不再影响上颚。

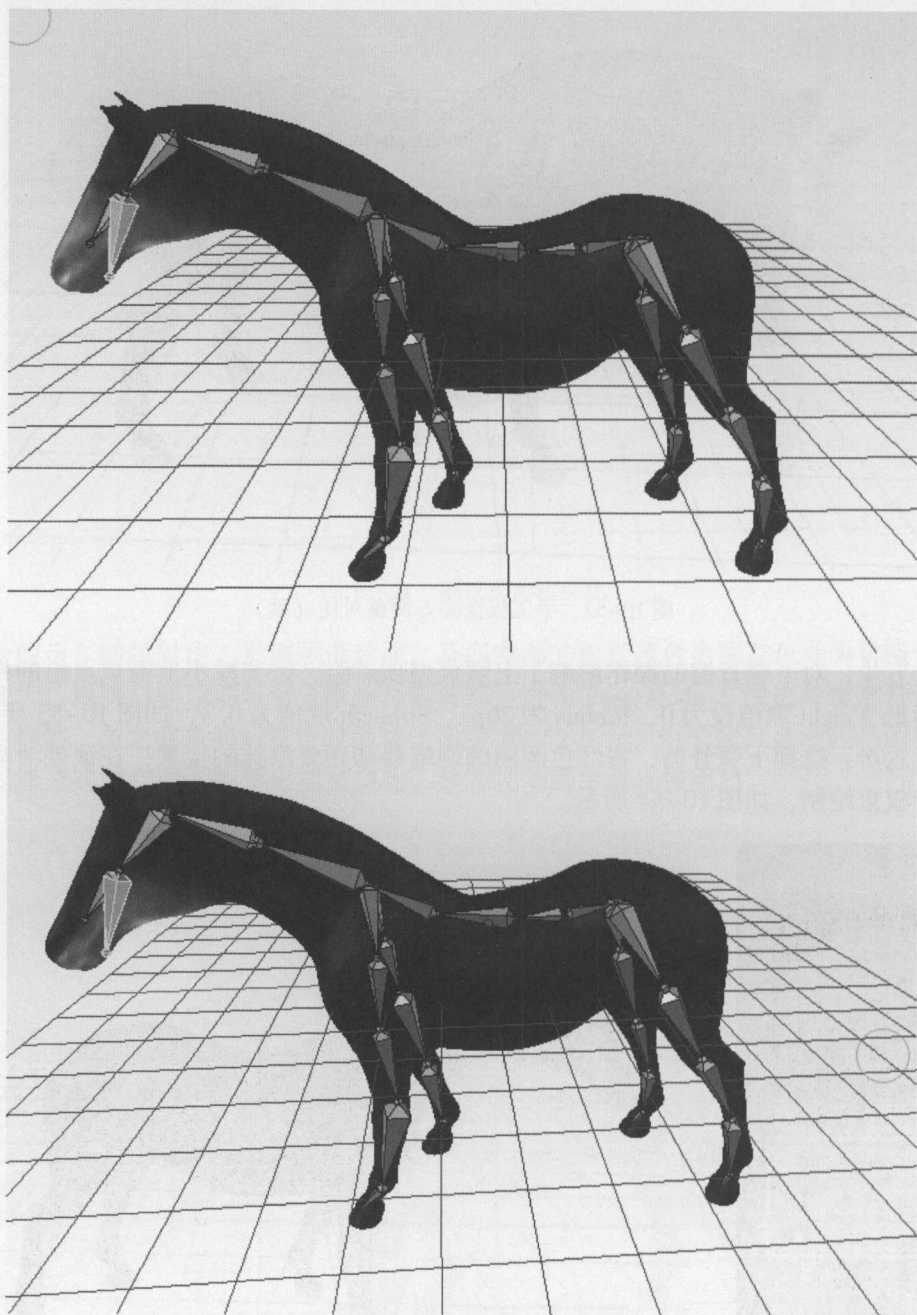


图 10-55 权重绘制结果对比

第十八步：选择下颚骨骼，在 Pose Mode 下按 R 键适当旋转，与之前未调整权重时的对比如图 10-56 和图 10-57 所示。通过对比发现，旋转下颚骨骼模型，上颚不再发生变形，

达到了预期的结果。对上颚的调整也可以采用同样的方法，这里就不再重复。

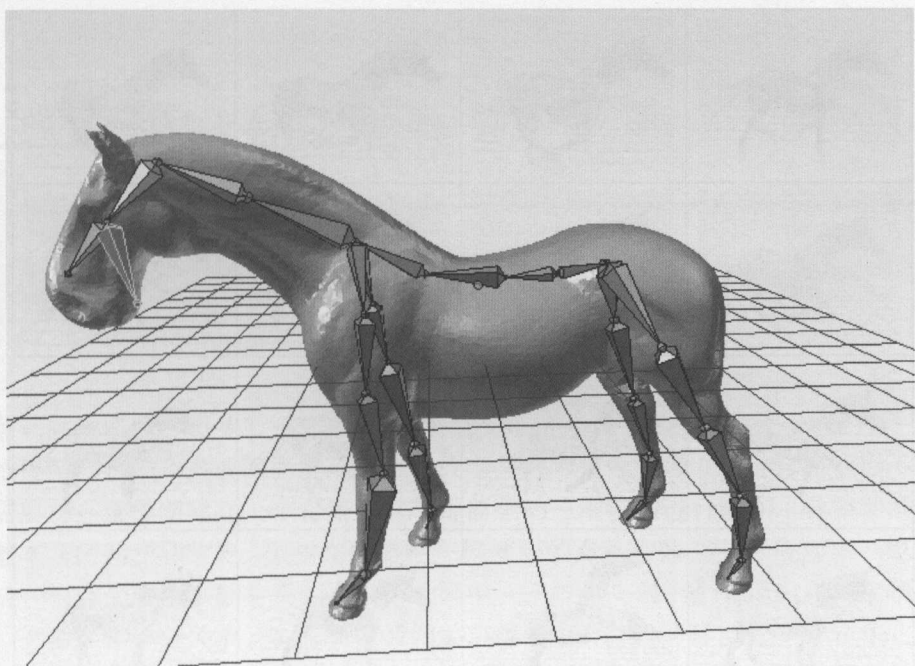


图 10-56 权重绘制前的姿态

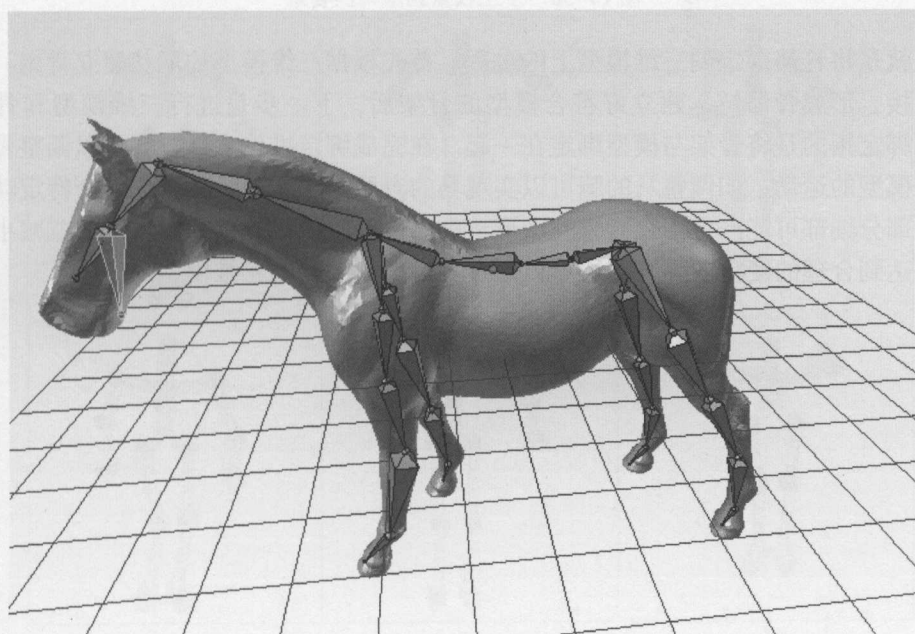


图 10-57 新的权重姿态调整

第十九步：调整马匹的姿态，查看其他部位有无不妥之处，按照上述方法依次修改其他部位的权重。

第二十步：用手工绘制的权重进行马匹姿态调整，效果如图 10-58 所示。

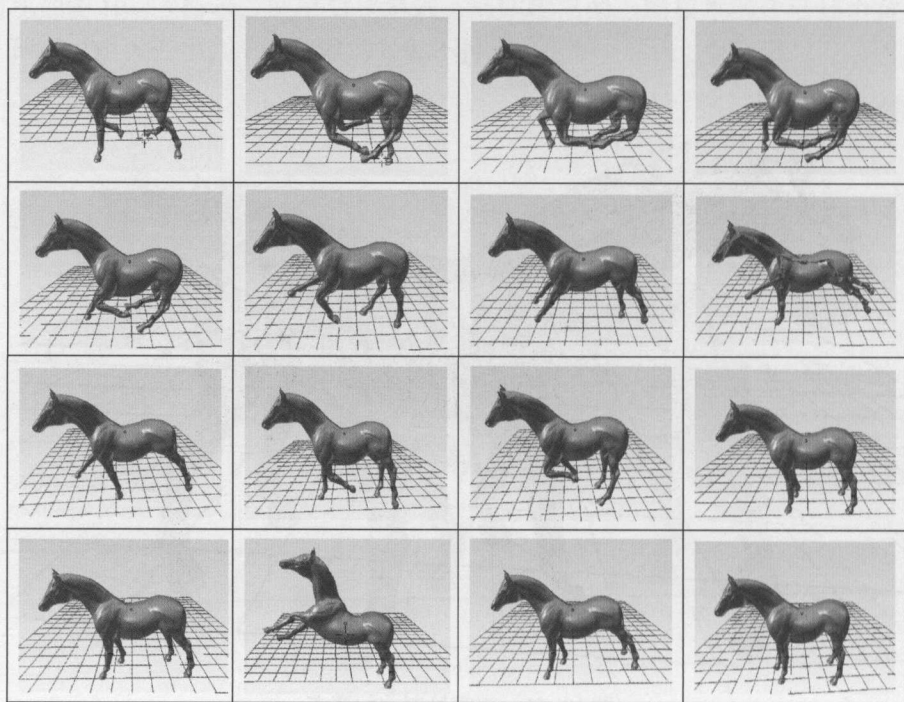


图 10-58 手工权重调整马匹姿态

蒙皮就是将骨骼绑定到三维模型上的过程。首先根据三维模型的形状建立骨架，骨架之间互相连接，形成骨骼链。建立好符合模型的骨架后，下一步是进行三维模型和骨架的绑定，骨架绑定指的是将骨架与模型绑定在一起。在完成绑定操作之后，就可以调整骨骼从而实现三维模型的运动，如调整马的腿可以实现马的奔跑。在调整骨架实现某个特定动作的时候，模型部分细部可能产生不合时宜的变形，这时候可以进行权重绘制，增大或减小某些顶点的权重达到合理的变化效果。从而可以通过骨骼的驱动驱动三维模型的变形。

第 11 章 骨骼动画算法



11.1 动作捕捉

骨骼动画是使用“骨头”使一个三维模型变形或者改变姿势，而不是通过手动编辑和移动三维模型的每个顶点或面而实现姿势的改变。与外包框变形、拉普拉斯变形不一样，骨骼动画需要对三维模型建立一个骨骼，这个骨骼驱动三维模型进行动作。骨骼动画分为两部分，骨骼本身的动作和蒙皮。蒙皮指的是把三维模型的点和骨骼一一对应起来，也就是每个顶点对应一根骨头或者多根骨头。在骨骼动画中，一根骨头或一个关节只是一组顶点的一个控制点。类似人体里的关节，如膝关节或腕关节等，当骨头运动时，每个附着在它上的顶点也跟着运动，同时某个骨头自身的运动也会导致其他骨头的运动。如图 11-1 所示的一个人体骨骼的各种不同动作。

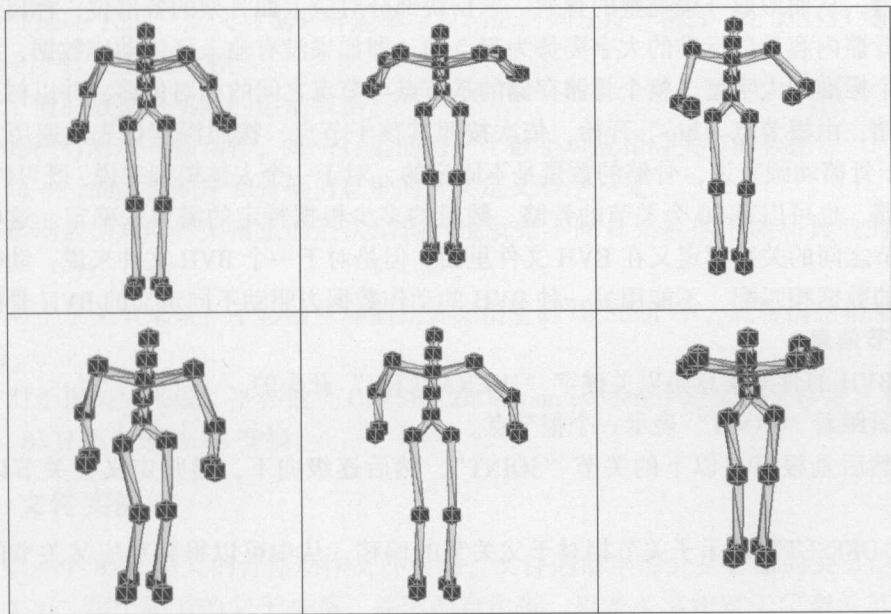


图 11-1 骨骼动画实例

骨骼动画中的动作可以用两种方法获得，一种是手工调节，也就是调整每个关节或者骨骼的位置和方向，从而得到一个特定的姿势。这种方法比较烦琐，很难得到一个满意的动作。另一种方法是通过动作捕捉设备对真实的人物运动进行捕捉，从而把得到的数据进行处

理，用这些数据驱动骨骼，从而使骨骼动起来，得到骨骼动画。这种方法能够得到很真实且自然的动作，比较方便。用户只要使用动作捕捉的设备，把想要的动作做出来即可。但是这种方法很难捕捉除了人物之外的其他动作，如奔跑的马、咆哮的老虎等。两种方法可以结合起来，人物的动作可以用动作捕捉设备进行捕捉，其他的动作可以用软件手工进行制作。

动作捕捉设备根据原理的不同可以分为机械式的、光学的、基于陀螺仪传感器的设备。光学的设备需要在若干个摄像头和人物身上的关节部位设置标记点，通过摄像头追踪计算这些标记点的运动，从而最终得到人物骨骼的动作数据。基于陀螺仪传感器的设备在人物的关节骨骼部位放置若干个陀螺仪，这些陀螺仪计算骨骼的运动，如平移、旋转等，从而得到骨骼最终的运动数据。这些动作捕捉的设备各有优缺点，可以根据需要来使用。捕捉出来的数据一般都具有噪音，需要用特定的软件（如 MotionBuilder）进行调整，从而得到最终的骨骼动作数据。



11.2 BVH 文件格式

11.2.1 BVH 格式定义

BVH (Biovision Hierarchical Data) 是一种骨骼动画数据的文件存储格式。动作捕捉设备捕捉出来的骨骼动作数据都存在文件里面，这种文件的格式一般都是 BVH 的格式。BVH 文件分为两个主要部分：骨架信息和数据块。骨架信息按照层级关系，定义了各个关节的位置和旋转分量，从而形成一个完整的骨架。数据块部分对应上面骨架的各部位，标出每帧的数据信息。骨骼内容是以标准的大字姿势为起点的，即如果没有输入任何动作数据，那么骨骼应该是一个标准的大字型。整个骨骼存储的是节点与节点之间的相对位移，并以树形结构进行文件存储，由根节点“hip”开始，依次按照其孩子节点，按照深度优先的遍历方式进行存储。对于骨骼动画来说，骨骼的数量是不固定的，对于一个人体模型来说，既可以有 30 个关节的骨骼，也可以有 60 个关节的骨骼。数量的多少根据特定的需要来确定。这些关节的数量和关节之间的关系都定义在 BVH 文件里面。但是对于一个 BVH 文件来说，动作的数据要和关节的数据相匹配，不能用另一种 BVH 的动作数据去驱动不同定义的 BVH 骨骼。

1. 关节信息

- (1) BVH 的骨架信息是以关键字“HIERARCHY”开头的。
- (2) 紧跟着“ROOT”表示一个根节点。
- (3) 然后是根节点以下的关节“JOINT”，然后逐级向下，递归定义父关节以下的子关节。
- (4) “OFFSET”表示子关节相对于父关节的偏移，从中可以得到对应父关节的长度和方向。
- (5) 当子关节不止一个时，采用第一个子关节的数据。
- (6) 接下来是“CHANNELS”，给出了关于 channel 的个数和名称，channel 表示这个关节的自由度数量。
- (7) ROOT 总是拥有 6 个 channels。
- (8) 而一般的 JOINT 只有 3 个，比 ROOT 缺少了 XYZ 的 position 信息。

(9) 原因是子关节只需要根据它相对于父关节的偏移就可以算出它在坐标系中的具体位置了。

(10) rotation channel 定义了动作存储的顺序, 一般是 Zrotation Xrotation Yrotation。

(11) BVH 格式运动数据的顺序不是 X、Y、Z 而是 Z、X、Y, 因此需要按照这个顺序处理数据。

(12) “End Site” 表示终结递归, 该关节的定义到此为止, 可看作一个终端标示器。

2. 关节实例

```
JOINT ltibia
{
    OFFSET 2.59720 -7.13576 0.00000
    CHANNELS 3 Zrotation Yrotation Xrotation
}
```

其中:

(1) Ltibia 为关节名称。

(2) OFFSET 2.59720 -7.13576 0.00000 为关节偏移量。

(3) CHANNELS 3 Zrotation Yrotation Xrotation 为通道顺序。

3. 数据块

(1) 数据块以 “MOTION” 关键字开始。

(2) “Frames” 定义帧数。

(3) “Frame Time” 定义数据采样速率 - 每帧的时间长度。例如 -0.033333 则表示 BVH 动画的采样速率为每秒 30 帧。

(4) 下面的数据是实际的运动数据。

(5) 每一行对应一帧动作, 每一行里的每一个数字对应相应关节的数据。

(6) 每一行的数据顺序和骨架信息里的骨架定义顺序一致。

(7) 运动数据对应骨架信息的层次结构, 也就是不同骨架数据不能和其他的运动数据混用。

(8) 对于子关节来说, 平移信息存储在骨架信息的 OFFSET 中, 旋转信息则来自于 MOTION 部分。

(9) 对于根关节来说, 平移量是 OFFSET 和 Motion section 中定义的平移量之和。

(10) BVH 不考虑 Scale 变换。

11.2.2 文件实例

下面是一个 BVH 格式的骨骼动作文件实例, 文件的前半部分是骨骼的关节信息, 定义了多少个关节、每个关节的父子关系、关节的自由度, 以及关节相对于父亲关节的偏移位置, 还有关节的名称信息。根据这部分的数据可以绘制出骨骼动画的骨骼信息。

文件的后半部分是骨骼动画的动作数据, 这些数据和上半部分骨骼定义的关节数据一一对应。每一行表示一个姿势, 把这些每一行的动作显示出来就构成一个完整的动画。

```
HIERARCHY
```

```

ROOT hip
{
    OFFSET 0.00000 0.00000 0.00000
    CHANNELS 6 Xposition Yposition Zposition Zrotation Yrotation Xrotation
    JOINT lhipjoint
    {
        OFFSET 0 0 0
        CHANNELS 3 Zrotation Yrotation Xrotation
        JOINT lfemur
        {
            OFFSET 1.65674 -1.80282 0.62477
            CHANNELS 3 Zrotation Yrotation Xrotation
            JOINT ltibia
            {
                OFFSET 2.59720 -7.13576 0.00000
                CHANNELS 3 Zrotation Yrotation Xrotation
                JOINT lfoot
                {
                    OFFSET 2.49236 -6.84770 0.00000
                    CHANNELS 3 Zrotation Yrotation Xrotation
                    JOINT ltoes
                    {
                        OFFSET 0.19704 -0.54136 2.14581
                        CHANNELS 3 Zrotation Yrotation Xrotation
                        End Site
                        {
                            OFFSET 0.00000 -0.00000 1.11249
                        }
                    }
                }
            }
        }
    }
}

JOINT rhipjoint
{
    .....
}

//其他后续的关节和前面的定义类似,此处省略。

```


MOTION

Frames:483

Frame Time:0.033333

```

9.4455 17.8610 - 0.5000 - 0.9193 - 4.1156 - 3.4650 0.0000 - 0.0000 0.0000
-22.2471.....
9.4469 17.8538 - 0.5182 - 0.9276 - 4.1650 - 3.2886 0.0000 - 0.0000 0.0000
-22.2333.....
9.4515 17.8423 - 0.5260 - 0.6609 - 4.1004 - 3.1654 0.0000 - 0.0000 0.0000
-22.5652.....
9.4570 17.8445 - 0.5267 - 0.0553 - 3.9244 - 3.1882 0.0000 - 0.0000 0.0000
-23.2700.....
9.4603 17.8475 -0.5365 0.4420 -3.7204 -3.1564 0.0000 -0.0000 0.0000 -23.8262.....
9.4644 17.8493 -0.5551 0.7344 -3.5714 -2.9579 0.0000 -0.0000 0.0000 -24.1497.....
9.4666 17.8582 -0.5849 0.4405 -3.5992 -2.5087 0.0000 -0.0000 0.0000 -23.7689.....
9.4642 17.8591 - 0.6213 - 0.1511 - 3.6158 - 1.9626 0.0000 - 0.0000 0.0000
-23.0073.....
.....
.....

```

如图 11-2 和图 11-3 展示了两个骨骼动画的截图，一个是踢腿动作，一个是跳跃动作。这些动作的数据都包含在 BVH 格式的下半部分。

1) 踢腿动作

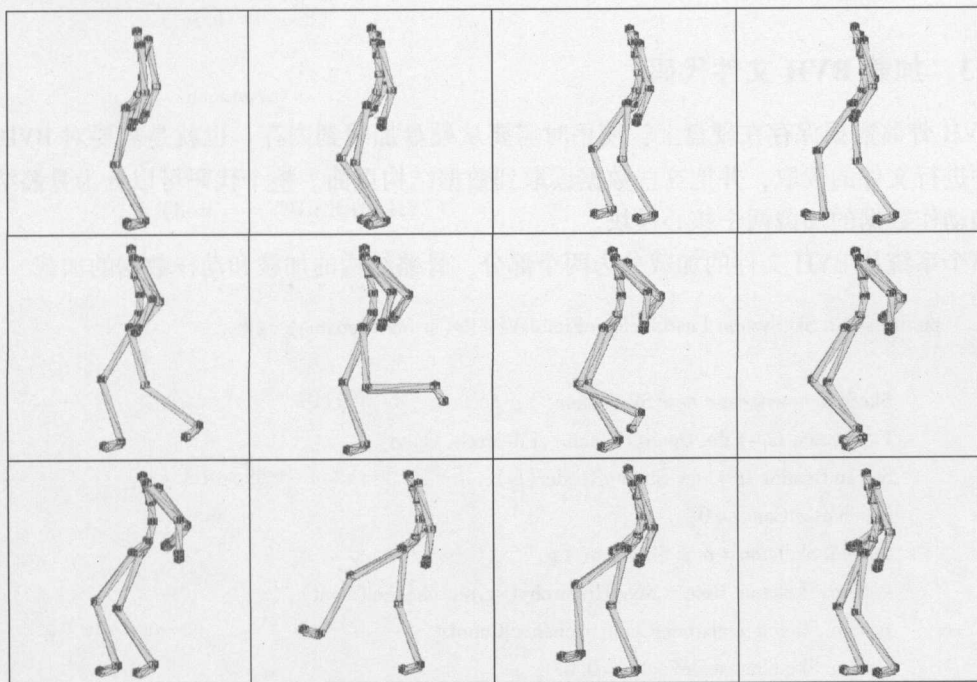


图 11-2 踢腿动作

2) 跳跃

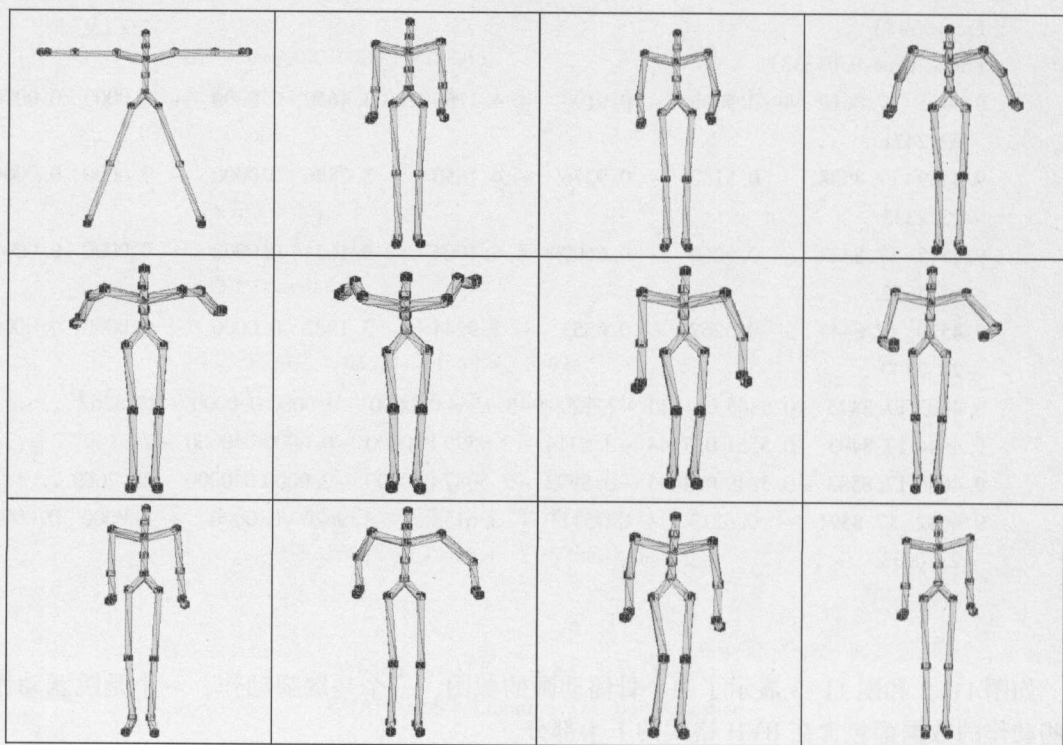


图 11-3 跳跃动作

11.2.3 加载 BVH 文件代码

BVH 骨骼数据保存在硬盘上，使用时需要从硬盘加载到内存。也就是需要对 BVH 格式的文件进行文件的读取，并把这些数据读取到数据结构里面。整个代码可以分为骨骼结构的加载和动作数据的加载两个核心模块。

整个系统从 BVH 文件的加载分为两个部分，骨骼结构的加载和动作数据的加载

```
public static SkeSystem LoadSkeletonFromBVHFile( string filename)
{
    SkeSystem system = new SkeSystem();
    FileStream fs = File. Open( filename, FileMode. Open );
    StreamReader sr = new StreamReader( fs );
    int chaneelCount = 0;
    system. Skeleton = new Skeleton();
    system. Skeleton. Root = ReadHierarchy( sr, out chaneelCount );
    system. Skeleton. channelCount = chaneelCount;
    system. Skeleton. scaleFactor = 0.03f;
    system. Motion = new AniMotion();
    int frames = 0;
```

```

double framesTime = 0;
system. Motion. MotionFrames = ReadMotionData(sr, out frames, out framesTime);
system. Motion. FrameCount = frames;
system. Motion. FrameTime = framesTime;
sr. Close();
return system;
}

```

1. 骨骼结构加载

```

private static SkeletonJoint ReadHierarchy( StreamReader sr, out int chaneelCount)
{
    Stack < SkeletonJoint > hiStack = new Stack < SkeletonJoint > ();
    SkeletonJoint root = null;
    bool isReadingHi = false;
    int nodeIndex = 0;
    String line = null;
    String preLine = null;
    chaneelCount = 0;
    while ( (line = sr. ReadLine()) != null)
    {
        String[] token = line. Split( new char[] { CHAR_TAB, CHAR_CR, CHAR_LF, CHAR_SPACE },
        StringSplitOptions. RemoveEmptyEntries);
        if ( token == null)
        {
            continue;
        }
        String head = token[0];
        if ( head == "HIERARCHY")
        {
            isReadingHi = true;
            continue;
        }
        if ( head == "MOTION")
        {
            isReadingHi = false;
            break;
        }
    }

    if ( isReadingHi)
    {
        if ( head == "{")
        {

```



```

        SkeletonJoint newNode = new SkeletonJoint();
        String[] lastLineToken = preLine.Split(new char[] { CHAR_TAB, CHAR_CR, CHAR_LF,
CHAR_SPACE },
        StringSplitOptions.RemoveEmptyEntries);

        if (lastLineToken[1] != "Site")
        {
            newNode.name = lastLineToken[1];
        }
        else
        {
            newNode.name = "";
        }
        nodeIndex++;
        if (lastLineToken[0] == "ROOT")
        {
            root = newNode;
        }
        else
        {
            hiStack.Peek().AddChild(newNode);
        }
        hiStack.Push(newNode);
    }

    if (head == "OFFSET")
    {
        SkeletonJoint currentNode = hiStack.Peek();
        currentNode.offset.x = double.Parse(token[1]) / scale_for_material;
        currentNode.offset.y = double.Parse(token[2]) / scale_for_material;
        currentNode.offset.z = double.Parse(token[3]) / scale_for_material;
    }

    if (head == "CHANNELS")
    {
        int channels = int.Parse(token[1]);
        SkeletonJoint currentNode = hiStack.Peek();
        for (int i = 2; i < 2 + channels; i++)
        {
            switch (token[i])
            {
                case "Xposition":

```

```

        //currentNode = chaneelCount;
        break;
    case "Yposition":
        //currentNode. yTranspotion = chaneelCount;
        break;
    case "Zposition":
        // currentNode. zTranspotion = chaneelCount;
        break;
    case "Zrotation":
        currentNode.ZRotationIndex = chaneelCount;
        break;
    case "Yrotation":
        currentNode.YRotationIndex = chaneelCount;
        break;
    case "Xrotation":
        currentNode.XRotationIndex = chaneelCount;
        break;
    default:
        break;
}

chaneelCount ++ ;
}
}
if (head == "}")
{
    if (hiStack.Count == 0)
        throw new Exception("There is a hierachy error!");
    hiStack.Pop();
}

}

preLine = line;
}
return root;
}

```

2. 动作数据的加载

```

private static double[ ][ ] ReadMotionData(StreamReader sr,
                                             out int frames,
                                             out double frameTime)
{

```

```

String line = null;
double[ ][ ] datas = null;
int currentFrame = 0;
frames = 0;
frameTime = 0;
while ( ( line = sr.ReadLine() ) != null )
{
    String[ ] tokens = line.Split(
        new char[ ] { CHAR_TAB, CHAR_CR, CHAR_LF, CHAR_SPACE },
        StringSplitOptions.RemoveEmptyEntries );
    if ( tokens == null )
    {
        continue;
    }
    if ( tokens[0] == "Frames:" )
    {
        frames = int.Parse(tokens[1]);
        datas = new double[frames][ ];
        continue;
    }
    if ( tokens[0] == "Frame" && tokens[1] == "Time:" )
    {
        frameTime = double.Parse(tokens[2]);
        continue;
    }
    datas[ currentFrame ] = new double[ tokens.Length ];
    for ( int i = 0; i < tokens.Length; i ++ )
    {
        datas[ currentFrame ][ i ] = double.Parse( tokens[ i ] );
    }
    currentFrame ++ ;
}
return datas;
}

```



11.3 骨骼结构算法

11.3.1 骨骼结构

在 BVH 文件数据加载到内存以后，需要使用这些数据驱动骨骼，并改变骨骼的姿势和形态。骨骼结构（Skeletal Structures）或者骨层级（Bone Hierarchies）指的是骨骼中各个关

节的连接关系。骨骼结构就是连续很多的骨头 (Bone) 相结合, 形成的骨层级。第一个骨头叫作根骨 (root bone), 是形成骨骼结构的关键点。其他所有的骨骼作为孩子骨 (child bone) 或者兄弟骨 (sibling bone) 附加在根骨之上。在骨骼动画中, 直接存储的数据是关节 (Joint) 的信息, 骨头的信息是从两个父子关节中间接计算出来的。动作的数据也是关节上的数据, 当关节的位置发生变化时, 骨头就可以根据父子关节的新位置计算出来。关节和骨头的关系如图 11-4 所示, 红色的是关节, 黄色的是骨头。

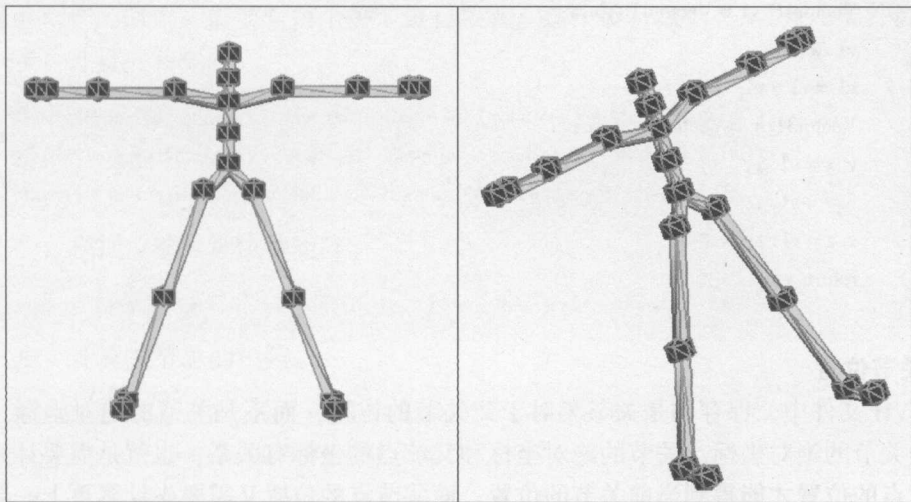


图 11-4 关节和骨头的关系

1. 根关节

根关节是一个模型中的终端关节。任何其他关节都以自己的路径最终关联到根关节。任何对根关节的操作, 如无论是平移或是旋转, 都会影响模型中的每个顶点。根关节是控制所有其他关节的关节。简单修改根关节, 能使角色直立行走, 或旋转从而倒过来在天花板上行走。在每个模型里只有一个根关节, 它没有父关节。根关节通常是许多骨头连接的地方, 如包括中部和下后部, 但没有明确要求根关节一定要在模型中的某个准确位置。每个模型都可以不同。

2. 父关节和子关节

一个关节可以有父关节和子关节。父关节的旋转和平移会影响所有子关节的位置。例如, 肘关节是手掌的父关节, 移动肘则影响手掌。在简单的骨骼动画里, 每个关节只有一个父关节。但是, 一个关节可以有許多子关节, 子关节是相对于父关节来说的。

11.3.2 算法原理

骨骼动画里, 最关键的是关节。每个节点的位置需要根据动作数据进行旋转位移得到最后的位置。在 BVH 骨架信息中, 可以得到每个关节相对于父关节的位移信息, 根据这个信息可以计算关节的初始位置。从 BVH 文件得到的运动数据是旋转角度和位移数据, 通过这些数据才能够得到关节的最终绝对坐标。骨骼动画的原来就是构建一个层次的骨骼, 骨骼由关节点构成, 根据关节点的绝对位置, 连接父子两个关节点就得到一个骨头。动作就是改变

这些关节的绝对位置，从而得到新的姿势。

1. 矩阵操作

通常需要把一个点通过矩阵改变位置。一个点是三维向量，需要先变为四维齐次坐标，然后乘以 4×4 的矩阵得到一个新的四维向量，最后再把四维向量变回三维坐标。

```
public Vector3D TransformOrigin( Matrix4D m)
{
    Vector4D v1 = Vector4D.Zero;
    v1.w = 1;
    v1 = v1 * m;
    Vector3D v = Vector3D.Zero;
    v.x = v1.x;
    v.y = v1.y;
    v.z = v1.z;
    return v;
}
```

2. 关节位置

在 BVH 文件中，保存的是关节相对于父关节的位移，而不是关节的绝对坐标，因此需要转变为关节的绝对坐标。关节的绝对坐标和父节点的坐标有关系，也就是需要计算出此关节的父节点的位置才能得到当前关节的位置。而父节点的位置又需要先计算更上一层的关节位置，以此类推直到根节点。因此每个关节需要保存一个矩阵，这个矩阵表示从根节点到当前节点的所有累计旋转位移矩阵。从而子节点计算的时候就不需要再从根节点开始，只需要得到父节点的矩阵就可以。但是在计算关节位置的时候，就不能随机进行，必须从根节点开始，层次往下进行计算，因为子关节的计算，需要父关节的矩阵，从而必须先计算父关节。

第一步：根据相对位移生成位移平移矩阵。

```
Matrix4D matrixTranslate = Matrix4D.Translation( this.scale * offset );
```

第二步：当前的位移平移矩阵乘以父节点的矩阵得到总的矩阵。

```
this.matrix = Matrix4D.Identity();
```

第三步：把第二步得到的矩阵乘以原点坐标得到关节的绝对坐标。

```
this.absolutePosition = TriMeshUtil.TransformOrigin( this.matrix );
```

第四步：实现。

```
public void ReferencePose()
{
    Matrix4D matrixTranslate = Matrix4D.Translation( this.scale * offset );
    this.matrix = Matrix4D.Identity();
    if ( this.parent != null )
    {
        this.matrix = matrixTranslate * parent.matrix;
    }
}
```

```

    }
    this.absolutePosition = TransformOrigin( this.matrix );
}

```

3. 关节旋转

关节位置得到的是每个关节没有任何旋转情况下的初始位置，一般是一个大字型的骨骼。在骨骼动作的时候，需要对每个关节进行旋转。这个旋转数据是从 BVH 文件读取的。

第一步：计算旋转矩阵。

```

Matrix4D matrixRotationX = Matrix4D. RotationX( rotation.x * Math.PI / 180 );
Matrix4D matrixRotationY = Matrix4D. RotationY( rotation.y * Math.PI / 180 );
Matrix4D matrixRotationZ = Matrix4D. RotationZ( rotation.z * Math.PI / 180 );

```

第二步：根据位移计算位移矩阵。

```

Matrix4D matrixTranslate = Matrix4D. Translation( this.scale * offset );

```

第三步：计算关节总的矩阵。

```

this.matrix = matrixRotationX * matrixRotationY * matrixRotationZ
              * matrixTranslate * parent.matrix;

```

第四步：计算关节旋转位移后的绝对位置。

```

this.absolutePosition = TransformOrigin( this.matrix );

```

第五步：代码实现。

```

public void Update( ref Vector3D rotation )
{
    Matrix4D matrixRotationX = Matrix4D. RotationX( rotation.x * Math.PI / 180 );
    Matrix4D matrixRotationY = Matrix4D. RotationY( rotation.y * Math.PI / 180 );
    Matrix4D matrixRotationZ = Matrix4D. RotationZ( rotation.z * Math.PI / 180 );
    Matrix4D matrixTranslate = Matrix4D. Translation( this.scale * offset );
    this.matrix = Matrix4D. Identity();
    this.matrix = matrixRotationX * matrixRotationY * matrixRotationZ
                  * matrixTranslate * parent.matrix;

    this.absolutePosition = TransformOrigin( this.matrix );
}

```

4. 根关节旋转

对于根关节来说，除了旋转，还有起始点的位移数据，也就是决定了模型起始点的位置。这个数据只有跟关节需要，因此和子关节生成绝对坐标的方法多一个位移矩阵。

第一步：生成起始点位移矩阵。

```

Matrix4D matrixStartPostion = Matrix4D. Translation( this.scale * translate );

```


第二步：生成根关节旋转矩阵。

```
Matrix4D matrixRotationX = Matrix4D. RotationX( rotation. x * Math. PI / 180 );
Matrix4D matrixRotationY = Matrix4D. RotationY( rotation. y * Math. PI / 180 );
Matrix4D matrixRotationZ = Matrix4D. RotationZ( rotation. z * Math. PI / 180 );
```

第三步：生成根关节位移矩阵。

```
Matrix4D matrixTranslate = Matrix4D. Translation( this. scale * offset );
```

第四步：生成总的矩阵。

```
this. matrix = matrixRotationX * matrixRotationY * matrixRotationZ
               * matrixTranslate * matrixStartPostion;
```

第五步：生成根关节位置坐标。

```
this. absolutePosition = TransformOrigin( this. matrix );
```

第六步：代码实现。

```
public void UpdateRoot( ref Vector3D rotation, ref Vector3D translate )
{
    Matrix4D matrixStartPostion = Matrix4D. Translation( this. scale * translate );
    Matrix4D matrixRotationX = Matrix4D. RotationX( rotation. x * Math. PI / 180 );
    Matrix4D matrixRotationY = Matrix4D. RotationY( rotation. y * Math. PI / 180 );
    Matrix4D matrixRotationZ = Matrix4D. RotationZ( rotation. z * Math. PI / 180 );
    Matrix4D matrixTranslate = Matrix4D. Translation( this. scale * offset );
    this. matrix = Matrix4D. Identity( );
    this. matrix = matrixRotationX * matrixRotationY * matrixRotationZ
                  * matrixTranslate * matrixStartPostion;
    this. absolutePosition = TransformOrigin( this. matrix );
}
```

5. 骨骼方向

在得到关节的位置和父关节的位置之后，此关节和父关节连接的骨骼方向可以根据两个位置计算得到。

```
public Vector3D BoneDirection
{
    get
    {
        if ( this. parent == null )
        {
            return Vector3D. Zero;
        }
        else
        {
            // ...
        }
    }
}
```

```

    {
        Vector3D BoneDirection = new Vector3D( this. absolutePosition. x
                                                - this. parent. absolutePosition. x,
                                                this. absolutePosition. y
                                                - this. parent. absolutePosition. y,
                                                this. absolutePosition. z
                                                - this. parent. absolutePosition. z );

        return BoneDirection;
    }
}

```

6. 遍历关节

虽然关节直接是层次结构，有父子关系。但是为了方便遍历关节，需要把所有关节放到一个列表里面。

```

public void RetrieveAllNode( SkeletonJoint root )
{
    int childCount = root. Children. Count;
    if ( childCount == 0 )
    {
        return;
    }
    root. Index = AllJoints. Count;
    AllJoints. Add( root );
    for ( int i = 0; i < childCount; i ++ )
    {
        RetrieveAllNode( root. Children[ i ] );
    }
}

```

7. 节点编号

```

public void BuildIndexTree( SkeletonJoint root )
{
    int childCount = root. Children. Count;
    if ( childCount == 0 )
    {
        return;
    }
    for ( int i = 0; i < childCount; i ++ )
    {
        if ( root. Children[ i ]. Name != "" )
        {

```

```

        root.ChildrenIndex.Add(root.Children[i].Index);
        root.Children[i].parentIndex = root.Index;
        BuildIndexTree(root.Children[i]);
    }
}

```

8. 动作数据

动作数据存储在一个二维数组里面，代码如下：

```
public double[,] MotionFrames = null;
```

整个动作的初始位置就是数组的前三个数；

```

public Vector3D StartPosition
{
    get
    {
        Vector3D startPosition = Vector3D.Zero;
        if (MotionFrames != null)
        {
            startPosition = new Vector3D(MotionFrames[0][0],
                                          MotionFrames[0][1],
                                          MotionFrames[0][2]);
        }

        return startPosition;
    }
    set
    {
        if (MotionFrames != null)
        {
            MotionFrames[0][0] = value.x;
            MotionFrames[0][1] = value.y;
            MotionFrames[0][2] = value.z;
        }
    }
}

```

数组的每一行对应一个姿势，每行的数据根据关节点里保存的序号对应到相应的关节，这些序号初始值是 -1，在 BVH 文件加载时候赋值。

```

public int XRotationIndex = -1;
public int YRotationIndex = -1;
public int ZRotationIndex = -1;

```


9. 动作更新

关节确定后, 需要从动作数据里读取数据, 更新关节的绝对位置。动作数据存储在一个二维数组里面。

第一步: 得到旋转数据。

```
double[] pose = poseInput.pose;
Vector3D rotation = new Vector3D(pose[joint.XRotationIndex],
                                   pose[joint.YRotationIndex],
                                   pose[joint.ZRotationIndex]);
```

第二步: 假如不是父节点, 只进行旋转。

```
if (joint.parent != null)
{
    joint.Update(ref rotation);
}
```

第三步: 对于父节点来说, 还需要进行位移。

```
Vector3D translate = new Vector3D(pose[0], pose[1], pose[2]);
translate = translate - startPos;
joint.UpdateRoot(ref rotation, ref translate);
```

第四步: 递归调用所有子节点。

```
while (child != null)
{
    if (child.leftMostChild != null)
    {
        SetUpPose(child, poseInput, startPos);
    }
    child = child.rightSibling;
}
```

第五步: 代码实现如下。

```
public void SetUpPose(SkeletonJoint joint, Pose poseInput,
                     Vector3D startPos)
{
    double[] pose = poseInput.pose;
    Vector3D rotation = new Vector3D(pose[joint.XRotationIndex],
                                      pose[joint.YRotationIndex],
                                      pose[joint.ZRotationIndex]);

    if (joint.parent != null)
    {
```

```

        joint.Update(ref rotation);
    }
    else
    {
        Vector3D translate = new Vector3D(pose[0], pose[1], pose[2]);
        translate = translate - startPos;
        joint.UpdateRoot(ref rotation, ref translate);
    }

    SkeletonJoint child = joint.leftMostChild;
    while (child != null)
    {
        if (child.leftMostChild != null)
        {
            SetUpPose(child, poseInput, startPos);
        }
        child = child.rightSibling;
    }
}

```

10. 生成关节模型

在得到关节的位置后，需要生成模型显示出来。也就是需要从关节的位置生成关节点的三维模型和骨头的三维模型。通常关节点是个球形，骨头是长方形。

第一步：在原点生成关节点模型。

```
TriMesh ball = BuildBall();
```

第二步：转移模型到关节点位置。

```

if (ball != null)
{
    ball.ModelName = joint.Name;
    SetUpBallColor(ball);
    TransformShapeJoint(ball, joint);
    skeletonShape.Add(ball);
}

```

第三步：在点生成骨骼模型。

```
TriMesh bone = BuildBone(joint.BoneLength);
```

第四步：转移模型到骨骼位置。

```

if (bone != null)
{
    bone.ModelName = joint.Name + ":" + joint.ParentName;
    SetUpBoneColor(bone);
}

```

```

TransformShapeBone( bone, joint );
skeletonShape. Add( bone );

```

第四步：代码实现。

```

public void CreateShapeJoints( SkeletonJoint joint )
{
    TriMesh ball = BuildBall( );
    if ( ball != null )
    {
        ball. ModelName = joint. Name;
        SetUpBallColor( ball );
        TransformShapeJoint( ball, joint );
        skeletonShape. Add( ball );
    }
    if ( joint. parent != null )
    {
        TriMesh bone = BuildBone( joint. BoneLength );
        if ( bone != null )
        {
            bone. ModelName = joint. Name + ":" + joint. ParentName;
            SetUpBoneColor( bone );
            TransformShapeBone( bone, joint );
            skeletonShape. Add( bone );
        }
    }
}

```

11. 生成骨骼模型

```

public List < TriMesh > skeletonShape = null;
public List < TriMesh > CreateShape( Skeleton skeleton )
{
    skeletonShape = new List < TriMesh > ( );
    for ( int i = 0; i < skeleton. AllJoints. Count; i ++ )
    {
        CreateShapeJoints( skeleton. AllJoints[ i ] );
    }

    for ( int i = 0; i < skeletonShape. Count; i ++ )
    {
        TriMeshUtil. SetUpNormalVertex( skeletonShape[ i ] );
    }
}

```



```
return skeletonShape;
```

12. 系统设计

整个骨骼系统分为如下几个类：

- (1) Skeleton。
- (2) SkeletonJoint。
- (3) AniMotion。
- (4) SysSkeleton。
- (5) Pose。
- (6) BVHFileParser。
- (7) SkeletonShapeBase。
- (8) SkeletonShapePoint。

其中，SkeletonJoint 类表示单个关节，包含在 Skeleton 类中。AniMotion 包含动作数据，SysSkeleton 包含 Skeleton 和 AniMotion 两个类。BVHFileParser 类负责解析 BVH 文件，并初始化 Skeleton 和 Animation 类。SkeletonShapeBase 类通过 Skeleton 类来生成三维形状。SkeletonShapePoint 类是 SkeletonShapeBase 类的子类，可以通过继承父类来生成不同的形状。例如，图 11-5 中生成的模型都不一样。其中后两个只有关节点的模型，而没有骨骼模型。

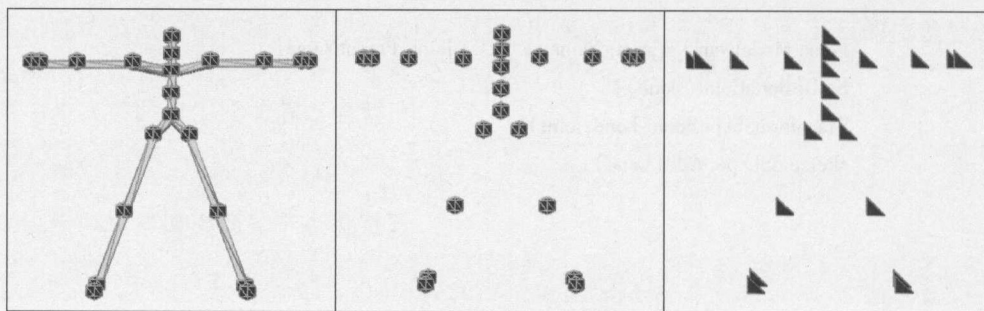


图 11-5 骨骼显示

第 12 章 蒙皮算法



12.1 概述

在骨骼的动作从文件加载到内存后，这些动作不仅仅可以驱动骨骼本身的变化和运动，还可以通过骨骼驱动和骨骼相连的三维模型进行变化和动作。用骨骼的动作改变三维模型的动作，就需要把三维模型和骨骼关联起来。这种关联的技术就是蒙皮（Skinning）技术。蒙皮把三维模型上的顶点和骨骼进行对应，每个顶点随着特定骨骼的位移、旋转进行相应的位移和旋转，从而骨骼的动作就改变了三维模型的形状。三维模型和骨骼的关联需要用三维软件（如 Maya、3DSMAX、Blender）进行。在三维模型和骨骼关联之后，这些数据可以保存到一个文件里面，通过读取文件里的蒙皮数据到内存，以及文件里骨骼的动作数据，两个数据相结合就可以使三维模型进行各种想要的动作。一般来说，这些数据可以保存为三个文件，第一个是动作文件，第二个是三维模型文件，第三个是使三维模型和骨骼对应的蒙皮数据的文件。也可以把这三种数据都包含到一个文件里面。经过蒙皮的三维模型除了原来三维模型文件格式所包含的顶点、面的信息之外，还需要有每个顶点和骨骼对应的信息。一般来说，一个顶点可以对应若干个骨骼，因此需要给每个对应的骨骼分配权重，从而确定这个顶点受到相对应的每个骨骼影响的大小。蒙皮数据里的骨骼是间接定义的，骨骼位置是由此骨骼两端的两个关节点位置得到的。骨骼的变换数据也是存储在关节点上的。因此三维模型的每个顶点和关节点直接相关联，而不是和两个关节点相连的骨骼进行相关联。

蒙皮算法的核心是设定权重，也就是决定三维模型上的每个顶点和相对应关节点的权重大小。这个设定权重的过程可以通过手工进行设置。一般来说，距离该顶点近的关节点权重数值比较大，距离该顶点远的关节点权重数值比较小。在权重确定之后，需要根据权重的数值，以及关节点的旋转位移改变与这个关节点相连的顶点的位置，有两张算法可以实现这个过程，分别是线性混合算法（Linear Blend Skinning）和对偶四元数算法（Dual Quaternion Skinning）算法。

如图 12-1 所示是一个女孩三维模型被骨骼驱动的蒙皮动画图示。其中红色线条表示骨骼，半透明模型表示的是和骨骼相关联的三维模型。随着红色骨骼的位置发生变化，三维模型也就发生了相应的改变。

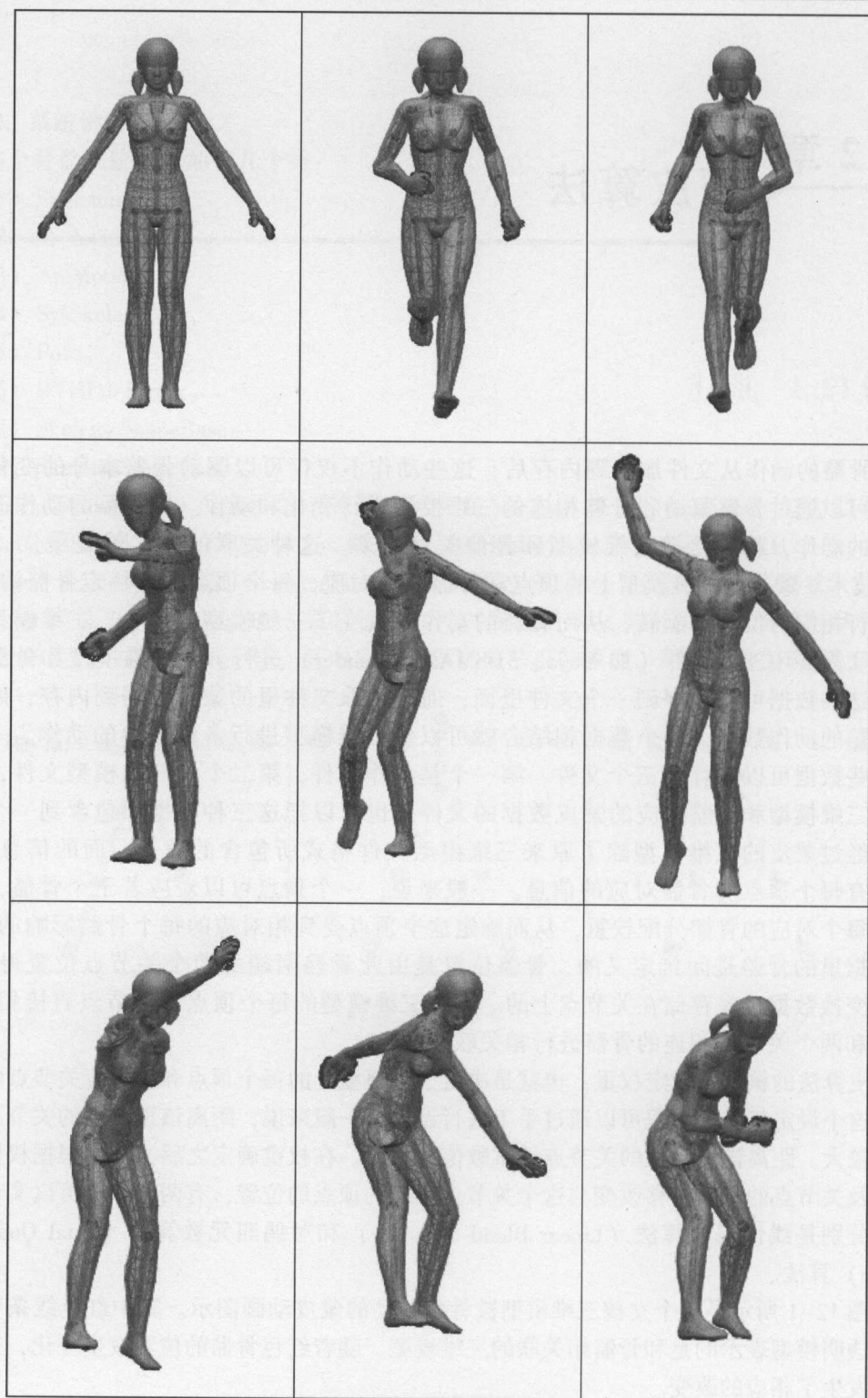


图 12-1 蒙皮动画图示



12.2 SMD 蒙皮文件

12.2.1 文件格式定义

蒙皮文件除了三维模型数据之外，还需要包含骨骼和权重数据。这些文件格式有很多，其中一种是 SMD 格式。SMD 是一种以 ASCII 码进行编码的文件格式。SMD 文件格式分为 Nodes、Skeleton、Triangles 三大模块。其中，Nodes 模块定义了骨骼的层次、每个关节的父子关系，每个 Node 相当一个关节；Skeleton 模块定义了动作数据的所有帧数，以及每一帧中每个关节的位置和旋转的角度；Triangles 模块定义了三维模型的顶点位置，以及和关节相连的权重。SMD 格式里面关节的层次和关节的位置分开进行定义。SMD 格式里的动作数据包含了每个关节的位移和旋转角度数据，一般来说，关节的位移数据在动作的过程中是保持不变的，从而可以保持骨骼的大小不变。但是在每一帧中存储位移的数据，就可以动态地改变三维模型骨骼的大小。

SMD 文件把蒙皮数据和骨骼的动作数据分为两个文件，其中蒙皮数据的文件包含三维模型、骨骼层次、权重信息。动作数据文件只包含动作的所有帧的数据。蒙皮文件中的 Skeleton 模块只有一帧数据，这一帧数据用来初始化骨骼的位置，而动作文件包含了完整的动作帧数据。动作文件不包含三维模型信息，需要把动作通过骨骼名与模型文件绑定。这样把蒙皮数据和动作数据分为两个文件，就可以灵活地把同样的动作应用到不同的三维模型上，以及对同样的三维模型应用不同的动作。下面是 SMD 文件的各个模块的详细格式介绍。

1. Nodes

这部分是骨骼的列表，语法如下。

```
nodes
```

节点块开始。

```
<int|ID> " <string|Bone Name> " <int|Parent ID>
```

骨骼定义。根节点 Parent ID 为 -1。SMD 通过骨骼名关键匹配不同文件中的骨骼。

```
end
```

块结束。

示例：

```
nodes
0 "root" -1
1 "child" 0
end
```

2. Skeleton

这部分是骨骼每帧的动作，至少要有一帧作为初始动作，语法如下。

```
skeleton
```

动作块开始。

```
time <int>
```

开始新的一帧。

```
<int|bone ID> <float|PosX PosY PosZ> <float|RotX RotY RotZ>
```

骨骼的动作。根节点的坐标为绝对坐标，其他节点为相对父节点的坐标。旋转用弧度单位的欧拉角表示。

```
end
```

块结束。

示例：

```
skeleton
time 0
0      0 0 0      1.570796 0 0
1      1 0 0      0 0 0
time 1
1      1 2 0      0 0 0
time 2
1      1 0 0      0 0 0
end
```

3. Triangles

这部分定义了模型的每个三角形的三个顶点，以及顶点上的 UV 和权重，语法如下。

```
triangles
```

三角形块开始。

```
<material>
```

开始一个新的三角形，以及三角形上的材质。材质名中的扩展名部分被忽略。

```
<int|Parent bone> <float|PosX PosY PosZ> <normal|NormX NormY NormZ> <normal|U V>
<int|links> <int|Bone ID> <normal|Weight> [...]
```

定义一个顶点。links 为与此顶点关联的骨骼个数，每个关联骨骼都有一个权重，权重之和为 1。

```
end
```

块结束。

示例：

```
triangles
my_material
```

```

0    0 0 0    0 0 1    0 1 1 0 1
0    0 -1 0    0 0 1    0 0 1 0 1
1    1 -1 0    0 0 1    1 0 1 1 1
my_material
0    0 0 0    1 0 1    0 1 1 0 1
1    1 -1 0    1 0 1    1 0 1 1 1
1    1 0 0    1 0 1    1 1 1 1 1
my_material
1    1 -1 0    0 0 1    1 0 1 1 1
0    0 -1 0    0 0 1    0 0 1 0 1
0    0 0 0    0 0 1    0 1 1 0 1
my_material
1    1 0 0    1 0 1    1 1 1 1 1
1    1 -1 0    1 0 1    1 0 1 1 1
0    0 0 0    1 0 1    0 1 1 0 1
end

```

12.2.2 文件加载

1. 内存数据结构

使用蒙皮技术进行三维模型的动画，首先需要把数据从文件加载到内存里面。因此，需要在内存里定义相应的数据结构。这些数据结构包括三维模型、权重、动画列表、骨骼层次、关节位置和旋转角度。

```

public TriMesh MeshCurrent = null;
public WeightPair[ ][ ] Weights;
public JointHierarchy[ ] Skeleton;
public Joint[ ] joints;
public List < Animation > AnimationList = new List < Animation > ( );
    public struct JointFrame
    {
        public Vector3D Position;
        public Quaternion Rotation;
    }

    public class Frame:List < JointFrame >
    {
        public int Time;
    }

    public class Animation:List < Frame >
    {

```



```

        public string Name;
    }

    public class JointHierarchy
    {
        public int Index;
        public string Name;
        public int Parent;
        public override string ToString()
        {
            return string.Format("{0},{2},{1}",
                this.Index, this.Name, this.Parent);
        }
    }

    public struct WeightPair
    {
        public int Joint;
        public double Weight;

        public WeightPair(int joint, double weight)
        {
            this.Joint = joint;
            this.Weight = weight;
        }
    }

```

2. 加载模型文件

这个函数把三维模型数据、权重数据、骨骼层次数据，以及骨骼的第一帧数据从蒙皮文件加载到内存里。

```

    public void LoadMesh(string file)
    {
        if (this.animations.Count != 0)
        {
            throw new Exception("不能换模型");
        }

        SMDNodesParser np = new SMDNodesParser();
        SMDSkeletonParser sp = new SMDSkeletonParser();
        SMDTrianglesParser tp = new SMDTrianglesParser();
        SMDParser parser = null;
        foreach (string[] token in SMDUtil.GetTokens(file))
        {

```

```

switch ( token[0] )
{
    case "nodes" :
        parser = np;
        parser.ReadBegin();
        break;
    case "skeleton" :
        parser = sp;
        parser.ReadBegin();
        break;
    case "triangles" :
        parser = tp;
        parser.ReadBegin();
        break;
    case "end" :
        parser.ReadEnd();
        parser = null;
        break;
    default :
        if ( parser != null )
        {
            parser.ReadLine(token);
        }
        break;
}

this.skeleton = np.Nodes;
Animation ani = sp.Animation;
ani.Name = "Init";
this.animations.Add(ani);
this.mesh = tp.Mesh;
this.weights = tp.Weights;
this.materials = tp.Materials;
this.name = SMDUtil.GetFileName(file);
}

```

3. 加载动画文件

这个函数把骨骼的动作数据从动作文件加载到内存里。

```

public void LoadAnimation(string file)
{
    if ( this.skeleton == null )
    {

```

```

        throw new Exception("必须先读入模型文件");
    }
    SMDNodesParser np = new SMDNodesParser();
    SMDSkeletonParser sp = new SMDSkeletonParser();
    SMDParser parser = null;
    foreach (string[] token in SMDUtil.GetTokens(file))
    {
        switch (token[0])
        {
            case "nodes":
                parser = np;
                parser.ReadBegin();
                break;
            case "skeleton":
                parser = sp;
                parser.ReadBegin();
                break;
            case "end":
                parser.ReadEnd();
                parser = null;
                break;
            default:
                if (parser != null)
                {
                    parser.ReadLine(token);
                }
                break;
        }
    }

    sp.Animation.Name = SMDUtil.GetFileName(file);
    Animation ani = this.Bind(sp.Animation, np.Nodes);
    this.animations.Add(ani);
}

```

4. 模型与动画绑定

因为动作数据和蒙皮数据是分开存储的，因此两个文件里加载的数据需要进行对应。可以通过骨骼名进行绑定，也就是两个文件里骨骼名称一样的骨骼就指的是同一个骨骼。

```

private Animation Bind(Animation ani, JointHierarchy[] nodes)
{
    NodeBinder binder = NodeBinder.Load(this.name, ani.Name);
    if (binder == null)
    {
    }
}

```



```

        binder = new NodeBinder( this.skeleton, nodes );
        binder.BindWithName();
        binder.modelName = this.name;
        binder.animationName = ani.Name;
        binder.Save();
    }

    Animation result = new Animation();
    result.Name = ani.Name;

    foreach ( var frame in ani )
    {
        Frame rf = new Frame();
        result.Add( rf );
        for ( int i = 0; i < binder.Count; i ++ )
        {
            if ( binder[ i ] == -1 )
            {
                rf.Add( this.animations[ 0 ][ 0 ][ i ] );
            }
            else
            {
                rf.Add( frame[ binder[ i ] ] );
            }
        }
        rf.TrimExcess();
    }
    result.TrimExcess();
    return result;
}

```

5. 文件解析器

每个 SMD 文件分为 3 个，因此需要 3 个解析器把文件中的每行转换成可用信息。

1) NodeParser

```

public class SMDNodesParser: SMDParser
{
    public JointHierarchy[] Nodes;

    private List<JointHierarchy> nodeList;

    public override void ReadBegin()
    {

```

```

        this. nodeList = new List < JointHierarchy > ( ) ;
    }

    public override void ReadLine( string[ ] token)
    {
        JointHierarchy joint = new JointHierarchy( ) ;
        joint. Index  = int. Parse( token[ 0 ] ) ;
        joint. Name  = token[ 1 ] ;
        joint. Parent = int. Parse( token[ 2 ] ) ;
        this. nodeList. Add( joint ) ;
    }

    public override void ReadEnd( )
    {
        this. Nodes = this. nodeList. ToArray( ) ;
    }
}

```

2) SkeletonParser

```

public class SMDSkeletonParser:SMDParser
{
    public Animation Animation;
    private Frame frame;
    public override void ReadBegin( )
    {
        this. Animation = new Animation( ) ;
    }
    public override void ReadLine( string[ ] token)
    {
        if ( token[ 0 ] == "time" )
        {
            this. frame = new Frame( ) ;
            this. Animation. Add( this. frame ) ;
        }
        else
        {
            int joint = int. Parse( token[ 0 ] ) ;
            double[ ] arr = new double[ 6 ] ;
            for ( int i = 0 ; i < 6 ; i ++ )
            {
                arr[ i ] = double. Parse( token[ i + 1 ] ) ;
            }
        }
    }
}

```

```

        this.frame.Add( new JointFrame()
        {
            Position = new Vector3D( arr,0 ),
            Rotation =
                Quaternion. RotationYawPitchRollRight( new Vector3D( arr,3 ) )
        } );
    }
}

```

3) TrianglesParser

这个函数读取 SMD 文件中的三维模型数据，生成内存 TriMesh 网格模型。

```

public class SMDTrianglesParser:SMDParser
{
    public TriMesh Mesh;
    public WeightPair[ ][ ] Weights;
    public string[ ] Materials;
    Dictionary < VertexKey, TriMesh. Vertex > dict;
    List < WeightPair[ ] > weightList;
    List < string > materialList;
    List < TriMesh. Vertex > face = new List < HalfEdgeMesh. Vertex > (3);

    public override void ReadBegin()
    {
        this.Mesh = new TriMesh();
        this.dict = new Dictionary < VertexKey, HalfEdgeMesh. Vertex > ();
        this.weightList = new List < WeightPair[ ] > ();
        this.materialList = new List < string > ();
    }

    public override void ReadLine( string[ ] token)
    {
        if ( token.Length == 1)
        {
            this.materialList.Add( token[0] );
            return;
        }
        double[ ] arr = new double[8];
        for ( int i = 0; i < 8; i ++ )
        {
            arr[i] = double.Parse( token[ i + 1 ] );
        }
    }
}

```



```

        Vector3D position = new Vector3D(arr, 0);
        Vector2D texCoord = new Vector2D(arr, 6);
        VertexKey key = new VertexKey(position, texCoord);
        if (dict.ContainsKey(key))
        {
            this.face.Add(this.dict[key]);
        }

        for (int i = 0; i < links; i++)
        {
            int bone = int.Parse(token[10 + i * 2]);
            double weight = double.Parse(token[11 + i * 2]);
            weightArr[i] = new WeightPair(bone, weight);
        }
        this.weightList.Add(weightArr);
    }

    if (this.face.Count == 3)
    {
        this.Mesh.Faces.AddTriangles(this.face.ToArray());
        this.face.Clear();
    }
}

else
{
    VertexTraits traits = new VertexTraits(position);
    traits.Normal = new Vector3D(arr, 3);
    traits.UV = texCoord;
    TriMesh.Vertex v = this.Mesh.Vertices.Add(traits);
    this.face.Add(v);
    this.dict.Add(key, v);

    int links = token.Length > 9 ? int.Parse(token[9]) : 0;
    WeightPair[] weightArr = new WeightPair[links];
    for (int i = 0; i < links; i++)
    {
        int bone = int.Parse(token[10 + i * 2]);
        double weight = double.Parse(token[11 + i * 2]);
        weightArr[i] = new WeightPair(bone, weight);
    }
    this.weightList.Add(weightArr);
}

```

```

    }

    if ( this. face. Count == 3 )
    {
        this. Mesh. Faces. AddTriangles( this. face. ToArray() );
        this. face. Clear();
    }
}

public override void ReadEnd()
{
    this. Mesh. Traits. BoundingBox =
        TriMeshUtil. ComputeBoundingBox( this. Mesh );
    this. Weights = weightList. ToArray();
    this. Materials = materialList. ToArray();
    this. dict. Clear();
    this. weightList. Clear();
    this. materialList. Clear();
}

struct VertexKey
{
    Vector3D Position;
    Vector2D TexCoord;

    public VertexKey( Vector3D position, Vector2D texCoord )
    {
        this. Position = position;
        this. TexCoord = texCoord;
    }
}
}

```

6. 示例

1) 简单动作 (见图 12-2)

以只有 3 个节点的骨骼为例, 分别在 x 轴上 -0.5 、 0 、 0.5 的位置上。

2) 跑步 (见图 12-3)

3) 挥手 (见图 12-4)



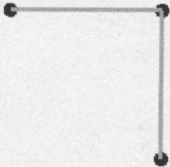
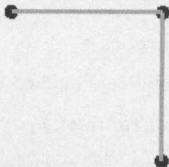
原始动作	根节点绕z轴旋转90°
	
中间结点根节点绕z轴旋转90°	末尾结点根节点绕z轴旋转90°，无效果
	

图 12-2 简单骨骼动画

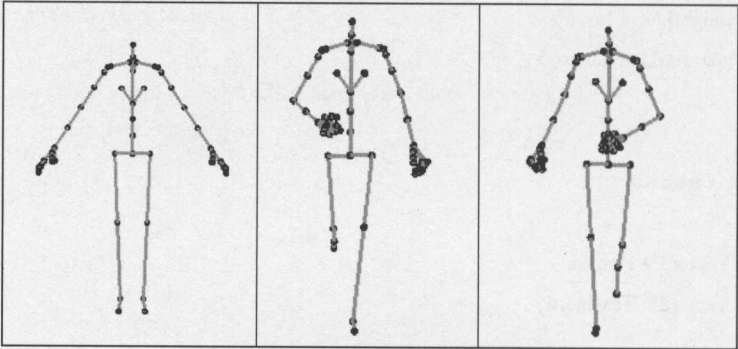


图 12-3 骨骼跑步动作

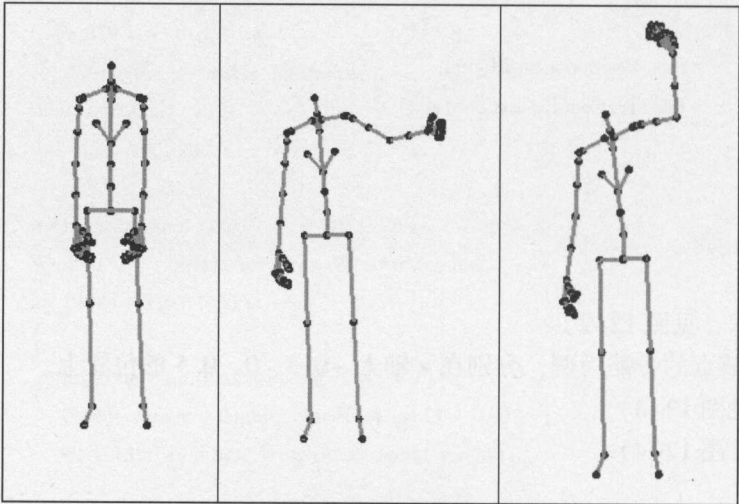


图 12-4 骨骼挥手动作



12.3 线性混合算法

在骨骼的变换和骨骼相对应的权重确定以后, 需要根据这些信息得到每个顶点的变换。最常用的方法是线性混合 (Linear Blender Skinning) 的方法。线性混合方法采用线性的方法从骨骼上关节的变换矩阵求得顶点的变换矩阵, 然后把每个顶点的位置变化由对应关节的变换矩阵加权求得。

1. 算法步骤

第一步: 关节 j_i 上保存有旋转数据 r_i 和偏移数据 t_i , 由 r_i 和 t_i 生成局部变换矩阵 M_i 。这个变换矩阵把一个顶点从坐标原点变换到关节当前的位置。

第二步: 若关节 j_i 为根节点, 那么该关节上的变换矩阵计算公式为 $B_i = M_i$ 。

第三步: 若关节 j_i 的父节点为 j_{ip} , 则该关节上的变换矩阵计算公式为 $B_i = M_i B_{ip}$ 。

第四步: 由于关节 j_i 上的数据随着动作的每一帧发生变化, 因此关节的变换矩阵每一帧都不一样。在初始帧时, 设置变换矩阵为 $A_i = B_i$ 。

第五步: 三维模型动作每帧都是由三维模型初始姿势变化而来的, 而不是在前一帧的基础上变化。所以需要每个关节 j_i 在初始帧时的变换矩阵 A_i 和在当前帧的变换矩阵 B_i 。

第六步: 关节所在位置是把坐标原点根据关节上数据表示的变换矩阵变换得到的。因此把关节所在的位置乘以变换矩阵的逆矩阵就得到坐标原点。如果已知关节第一帧的位置和变换矩阵, 那么关节在当前帧的位置可以有两种方法得到。第一种方法是从原点变换, 第二种方法根据关节在第一帧时的位置, 第一帧的变换矩阵的逆矩阵, 以及当前帧的变换矩阵得到。公式为 $J' = J \cdot A_i^{-1} B_i$ 。

第七步: 因此可以对每个关节 j_i 定义蒙皮变换矩阵为 $S_i = A_i^{-1} B_i$ 。

第八步: 对三维模型上的顶点 v , 坐标表示为 p , 三维模型在初始帧上的每个顶点坐标 p 是已知的数据和每一帧上关节的变换矩阵, 未知的数据是每个顶点在每一帧的变化数据。

第九步: 顶点坐标 p 右乘变换矩阵 S_i 的几何意义是把初始帧 p 和关节 j_i 的相对关系变换到当前帧的关节 j_i 上。也就是三维模型每个顶点在每一帧上的变换可以由如 $p' = p S_i$ 计算得到。

第十步: 但是三维模型上每个顶点受到多余一个关节的影响, 如果 w_i 表示关节 j_i 在此顶点 v 上的权重, 其中 $\sum w_i = 1, 0 \leq w_i \leq 1$ 。那么此顶点 v 的线性混合矩阵 $L = \sum w_i S_i$ 。

第十一步: 顶点 v 在当前帧的坐标计算公式为 $p' = p L$ 。

第十二步: 以上公式可以合并为 $p' = p \sum w_i A_i^{-1} B_i$, 从而求得每一帧上三维模型每个顶点变化后的位置。

2. 代码

第一步: 加载数据到内存。

由于每一帧三维模型都是从初始三维模型变换得到的, 因此需要对初始模型进行备份。

```
public SMDSsystem( TriMesh mesh, WeightPair[ ][ ] weights,
    JointHierarchy[ ] skeleton, List < Animation > animations)
{

    this. MeshReference = mesh;
    this. MeshCurrent = TriMeshIO. Clone( mesh );
    TriMeshUtil. SetUpNormalVertex( this. MeshCurrent );

    this. Weights = weights;
    this. Skeleton = skeleton;
    this. AnimationList = animations;
    this. InitJoint( );
}
```

第二步：计算每个关节的变换矩阵。

```
public void UpdateJoints( )
{
    for ( int i = 0; i < this. joints. Length; i ++ )
    {
        Matrix4D matLocal = Matrix4D. RotationQuaternion( this. joints[ i ]. Rotation );
        matLocal. Row4 = new Vector4D( this. joints[ i ]. Position, 1d );

        if ( this. Skeleton[ i ]. Parent == - 1 )
        {
            this. joints[ i ]. matGlobal = matLocal;
        }
        else
        {
            this. joints[ i ]. matGlobal = matLocal *
                this. joints[ this. Skeleton[ i ]. Parent ]. matGlobal;
        }
    }
}
```

第三步：初始化每个关节的变换矩阵，并设置第一帧变换矩阵。

```
public void InitJoint( )
{
    if ( this. AnimationList. Count > 0 )
    {
        this. curAni = 0;
        this. joints = new Joint[ this. Skeleton. Length ];
    }
}
```

```

for (int i=0;i < this.joints.Length;i++)
{
    this.joints[i] = new Joint();
    this.joints[i].Position = this.AnimationList[0][0][i].Position;
    this.joints[i].Rotation = this.AnimationList[0][0][i].Rotation;
}
this.UpdateJoints();
for (int i=0;i < this.joints.Length;i++)
{
    this.joints[i].matGlobalInit = this.joints[i].matGlobal;
}
}

```

第四步：计算骨骼上每个关节在当前帧的变换矩阵。

```

Matrix4D[] arr = new Matrix4D[ this.joints.Length ];
for (int i=0;i < arr.Length;i++)
{
    arr[i] = this.joints[i].matGlobalInit.Inverse()
        * this.joints[i].matGlobal;
}

```

第五步：计算每个顶点的线性混合权重。

```

TriMesh.Vertex coreV = this.MeshReference.Vertices[ v.Index ];
Matrix4D m = new Matrix4D();
foreach (var item in this.Weights[ v.Index ])
{
    Joint joint = this.joints[ item.Joint ];
    m += arr[ item.Joint ] * item.Weight;
}

```

第六步：根据公式计算当前帧三维模型每个顶点变换后的位置。

```

v.Traits.Position =
    TriMeshUtil.TransformationVector3D( coreV.Traits.Position, m );

public static Vector3D TransformationVector3D( Vector3D v, Matrix4D m )
{
    Vector4D v1 = new Vector4D();
    v1.x = v.x;
    v1.y = v.y;
    v1.z = v.z;
}

```



```

v1.w = 1;
v1 *= m;
v.x = v1.x;
v.y = v1.y;
v.z = v1.z;
return v;

```

3. 完整代码

```

public virtual void LBS()
{
    Matrix4D[] arr = new Matrix4D[ this.joints.Length ];
    for (int i = 0; i < arr.Length; i++)
    {
        arr[i] = this.joints[i].matGlobalInit.Inverse()
            * this.joints[i].matGlobal;
    }

    foreach (var face in this.MeshCurrent.Faces)
    {
        foreach (var v in face.Vertices)
        {
            TriMesh.Vertex coreV = this.MeshReference.Vertices[v.Index];
            Matrix4D m = new Matrix4D();
            foreach (var item in this.Weights[v.Index])
            {
                Joint joint = this.joints[item.Joint];
                m += arr[item.Joint] * item.Weight;
            }
            v.Traits.Position =
                TriMeshUtil.TransformationVector3D( coreV.Traits.Position, m );
        }
    }
}

```

4. 图示

图 12-5 是一个 LBS 算法踢腿动作的过程图示。



图 12-5 LBS 算法踢腿动作图示



12.4 对偶四元素数算法

12.4.1 数学概念

1. 对偶数

对偶数是复数的一种扩展,由实数单位 1 和对偶单位 ε 组成, ε 称为克利福德算符

$$\hat{a} = a + a'\varepsilon, \text{ 其中 } \varepsilon^2 = 0$$

运算法则

$$\hat{a} + \hat{b} = a + b + \varepsilon(a' + b')$$

$$\hat{a}\hat{b} = ab + \varepsilon(b'a + a'b)$$

2. 四元数

四元数是复数的一种扩展，由一个实部和 i, j, k 三个虚部组成。

$$q = w + xi + yj + zk$$

其中

$$i^2 = j^2 = k^2 = -1$$

$$ij = k, ji = -k$$

$$jk = i, kj = -i$$

$$ki = j, ik = -j$$

一般把四元数表示为一个标量和一个向量的组合

$$q = (w, v), v = (x, y, z)$$

运算法则

$$q_1 + q_2 = (w_1 + w_2, v_1 + v_2)$$

$$q_1 q_2 = (w_1 w_2 - v_1 v_2, w_1 v_2 + w_2 v_1 + (v_1 \times v_2))$$

$$q^* = (w, -v)$$

3. 对偶四元数

把对偶数中的实数替换为四元数可以得到对偶四元数。一个对偶四元数 $\hat{a} = a + a'\varepsilon$ 可以同时表示旋转和位移

$$a = r$$

$$a' = \frac{1}{2}tr$$

矩阵转化为对偶四元数，可以先把矩阵分解为旋转四元数和位移，再利用上式得出对偶四元数。类似地，也可以把对偶四元数转化为矩阵。

$$r = a$$

$$t = 2a'a^*$$

其中， r 是表示旋转的单位四元数， t 是以四元数表示的位移向量 $t = (0, \vec{t})$ 。

加法运算法则

$$\hat{a} + \hat{b} = a + b + \varepsilon(a' + b') = r_a + r_b + \frac{1}{2}\varepsilon(t_a r_a + t_b r_b)$$

对偶四元数相加时，旋转部分直接相加。

12.4.2 算法原理

对偶四元数蒙皮算法 (Dual Quaternion Skinning) 简称 DQS 算法。对偶四元数方法也需要计算关节的变换矩阵，但是不采用线性混合的方法来得到每个顶点最终的变换矩阵。而是先把关节上的变换矩阵变为对偶四元数，再把每个顶点对应的各个关节上的对偶四元数进行加权求和。然后再把求和的对偶四元数变换到矩阵，最后用得到的矩阵对顶点进行变换。DQS 算法通过对偶四元数间接得到的每个顶点变换矩阵和 LBS 算法直接求得的变换矩阵是不一样的。DQS 算法利用对偶四元数相加时旋转部分直接相加，从而可以解决 LBS 算法在关节运动时模型体积减小的问题。这种算法和 LBS 算法相比能够获得更好的变形效果。

1. 算法代码

1) 对偶四元数

对偶四元数代码主要模块是对偶四元数的求和、把矩阵变换为对偶四元数、把对偶四元数变换为矩阵。用对偶四元数对顶点进行变换：

```
public struct DualQuaternion
{
    public Quaternion A,B;
    public DualQuaternion( Quaternion q,Vector3D v)
    {
        this.A = q;
        Quaternion t = new Quaternion(v,0);
        this.B = 0.5 * t * q;
    }
    public DualQuaternion( Matrix4D m)
        :this( Quaternion. RotationMatrix( m),new Vector3D( m. Row4. ToArray( ),0))
    {
    }
    public void Sum( DualQuaternion other,double weight)
    {
        if ( Quaternion. Dot( this. A,other. A) <0)
        {
            other. A = - other. A;
            other. B = - other. B;
        }
        this. A += other. A * weight;
        this. B += other. B * weight;
    }
    public void Normalize()
    {
        double len = this. A. Length;
        this. A /= len;
        this. B /= len;
        this. B -= this. A * Quaternion. Dot( this. A,this. B);
    }
    public Vector3D TransformPointWithMatrix( Vector3D pos)
    {
        return TriMeshUtil. TransformationVector3D( pos,this. ToMatrix( ));
    }

    public Vector3D TransformPoint( Vector3D pos)
    {
    }
}
```

```

        Vector3D axyz = new Vector3D( this. A. ToArray(), 0 );
        Vector3D bxyz = new Vector3D( this. B. ToArray(), 0 );
        return pos + ( axyz. Cross( axyz. Cross( pos ) + pos * this. A. w + bxyz )
            + bxyz * this. A. w - axyz * this. B. w ) * 2;

    }

    public Matrix4D ToMatrix()
    {
        Matrix4D m = Matrix4D. RotationQuaternion( this. A );
        m. Row4 = 2 * new Vector4D( ( this. B * ( this. A. Conjugate() ) ). ToArray() );
        return m;
    }
}

```

2) 变形

第一步：初始化每个关节的变换矩阵。这一步的代码和 LBS 算法一样。

第二步：根据每个关节的 j_i 上的蒙皮矩阵 $S_i = A_i^{-1} B_i$ ，通过 S_i 生成 j_i 上的对偶四元数 D_i 。

```

DualQuaternion[] arr = new DualQuaternion[ this. joints. Length ];
for ( int i = 0; i < arr. Length; i ++ )
{
    arr[ i ] = new DualQuaternion( this. joints[ i ]. matGlobalInit. Inverse() *
        this. joints[ i ]. matGlobal );
}

```

第三步：对每个模型上的顶点 v ，坐标为 p ， w_i 表示关节 j_i 在顶点 v 上的权重，其中 $\sum w_i = 1, 0 \leq w_i \leq 1$ 。顶点 v 的对偶四元数 $D = \sum w_i D_i$ 。

```

TriMesh. Vertex coreV = this. MeshReference. Vertices[ v. Index ];
DualQuaternion sum = new DualQuaternion();
foreach ( var item in this. Weights[ v. Index ] )
{
    Joint joint = this. joints[ item. Joint ];
    sum. Sum( arr[ item. Joint ], item. Weight );
}
sum. Normalize();

```

第四步：把 D 转化为矩阵形式 M ，顶点 v 在当前帧的坐标 $p' = pM$ 。使用 DQS 方法更新顶点位置。

```

v. Traits. Position = sum. ApplyToPoint( coreV. Traits. Position );

```

2. 完整代码

```

public virtual void DQS()

```

```

{
    DualQuaternion[] arr = new DualQuaternion[ this.joints.Length ];
    for ( int i = 0; i < arr.Length; i ++ )
    {
        arr[i] = new DualQuaternion( this.joints[i].matGlobalInit.Inverse() *
            this.joints[i].matGlobal );
    }

    foreach ( var face in this.MeshCurrent.Faces )
    {
        foreach ( var v in face.Vertices )
        {
            TriMesh.Vertex coreV = this.MeshReference.Vertices[ v.Index ];
            DualQuaternion sum = new DualQuaternion();
            foreach ( var item in this.Weights[ v.Index ] )
            {
                Joint joint = this.joints[ item.Joint ];
                sum.Sum( arr[ item.Joint ], item.Weight );
            }
            sum.Normalize();
            v.Traits.Position = sum.TransformPointWithMatrix( coreV.Traits.Position );
        }
    }
}

```

3. 图示

图 12-6 是三维人物模型欢呼动作的图示。

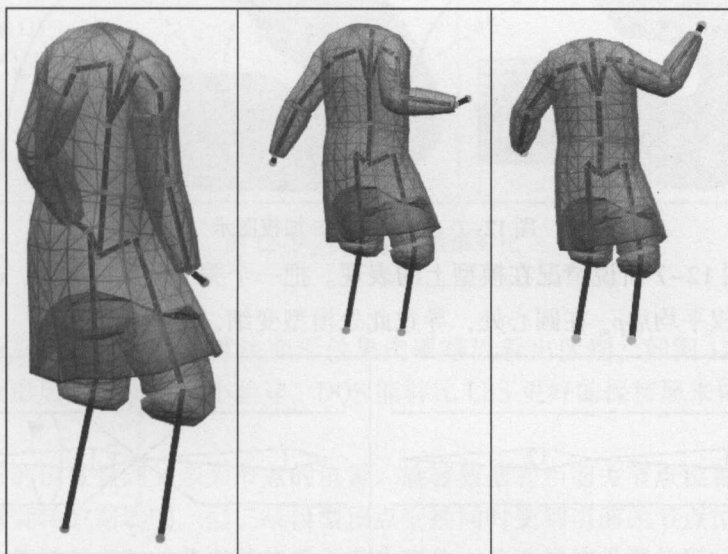


图 12-6 DQS 算法欢呼动作图示

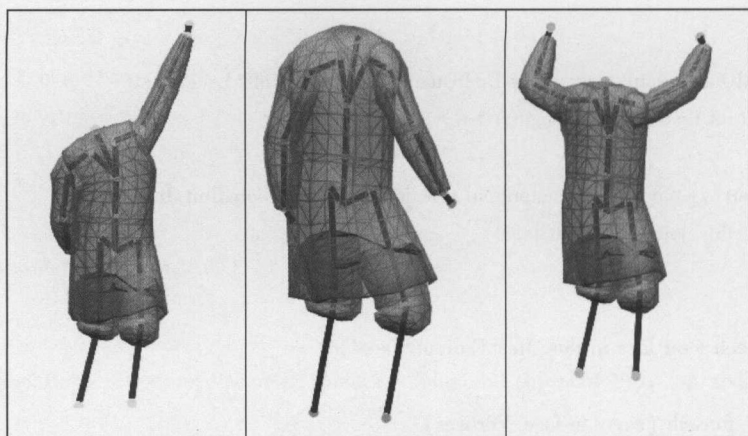


图 12-6 DQS 算法欢呼动作图示（续）



12.5 DQS 和 LBS 对比

12.5.1 优劣性

DQS 算法和 LBS 算法相比的优点是三维模型变形后体积变化小，而 LBS 算法经常会损失体积。这是由于两种算法加权求和的方式不同，从而每个顶点得到不同的变换矩阵，而 DQS 算法得到的变换矩阵能够保持三维模型原来的体积。假设以 p_1 和 p_2 权重均为 0.5 为例， \bar{p}_m 为加权结果。图 12-7 (a) 为 LBS，图 12-7 (b) 为 DQS。图 12-7 (a) 中 \bar{p}_m 相比 p_1 和 p_2 离圆心距离缩短，体现在模型上就是体积变小。图 12-7 (b) 中距离不变。

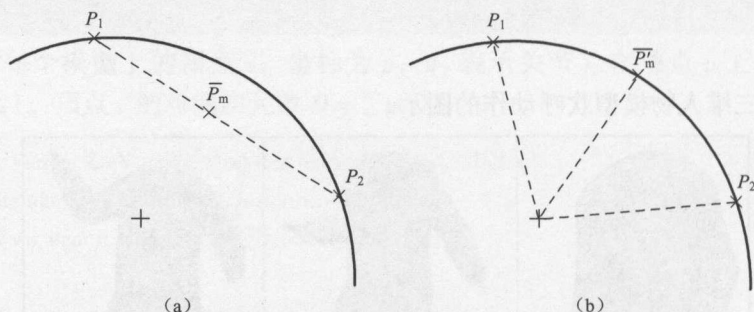


图 12-7 DQS 和 LBS 加权图示

图 12-8 为图 12-7 所说情况在模型上的表现。把一个关节绕轴旋转 π ， v_1 移动到了另一侧 v_2 的位置，加权平均后 \bar{p}_m 在圆心处，导致此处模型变细，体积缩小。

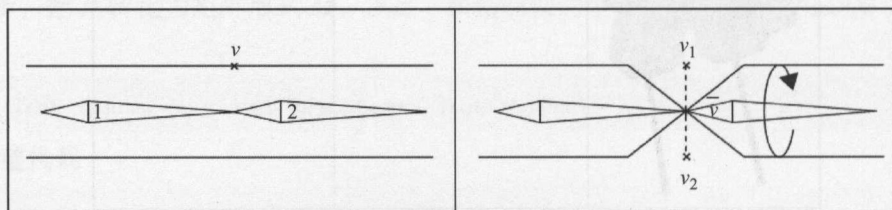


图 12-8 DQS 和 LBS 对比图示

1. 实验一

在通常的三维模型上，两种方法算法区别不明显，计算体积有微小差别。在特殊模型上，两种算法变形后的体积差异较大。图 12-9 是圆柱三维模型在 LBS 和 DQS 两种算法上的变形结果。从中可以看出旋转的时候，DQS 算法能够保持原来体积不变，而 LBS 算法就在弯曲的地方减少了三维模型的体积，从而造成不自然的变形效果。

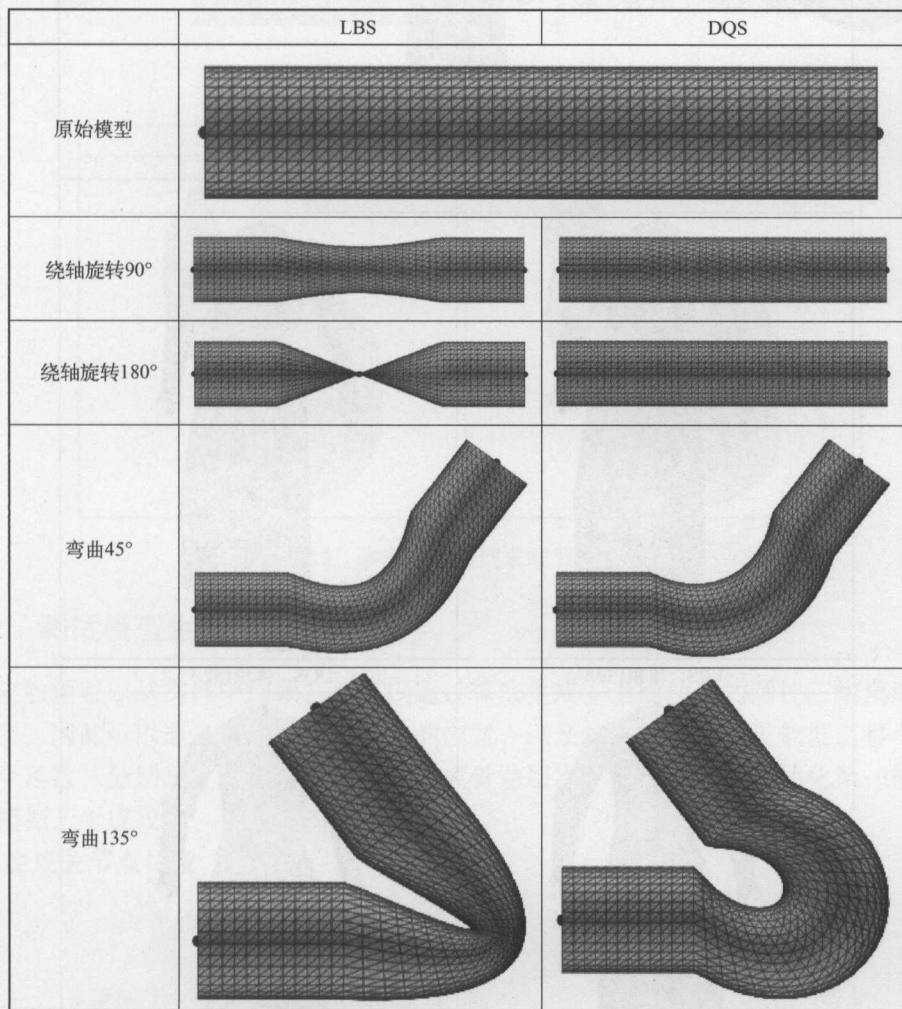


图 12-9 圆柱变形效果对比

2. 实验二

对于大部分三维模型，两种算法变形效果肉眼难以看出区别，如图 12-10 所示，可以从数据上看出变形后的体积有微小差异。DQS 能够比 LBS 更好的保持原来体积不变。

3. 实验三

每个关节的位置受到父亲关节的位置、旋转数据和当前关节位置数据的影响，但不受当前关节旋转数据影响，但三维模型顶点变换同时受到当前关节位置数据和旋转数据的影响。因此旋转节点时关节的位置不发生变化，从而整体骨骼的形状不发生变化，但

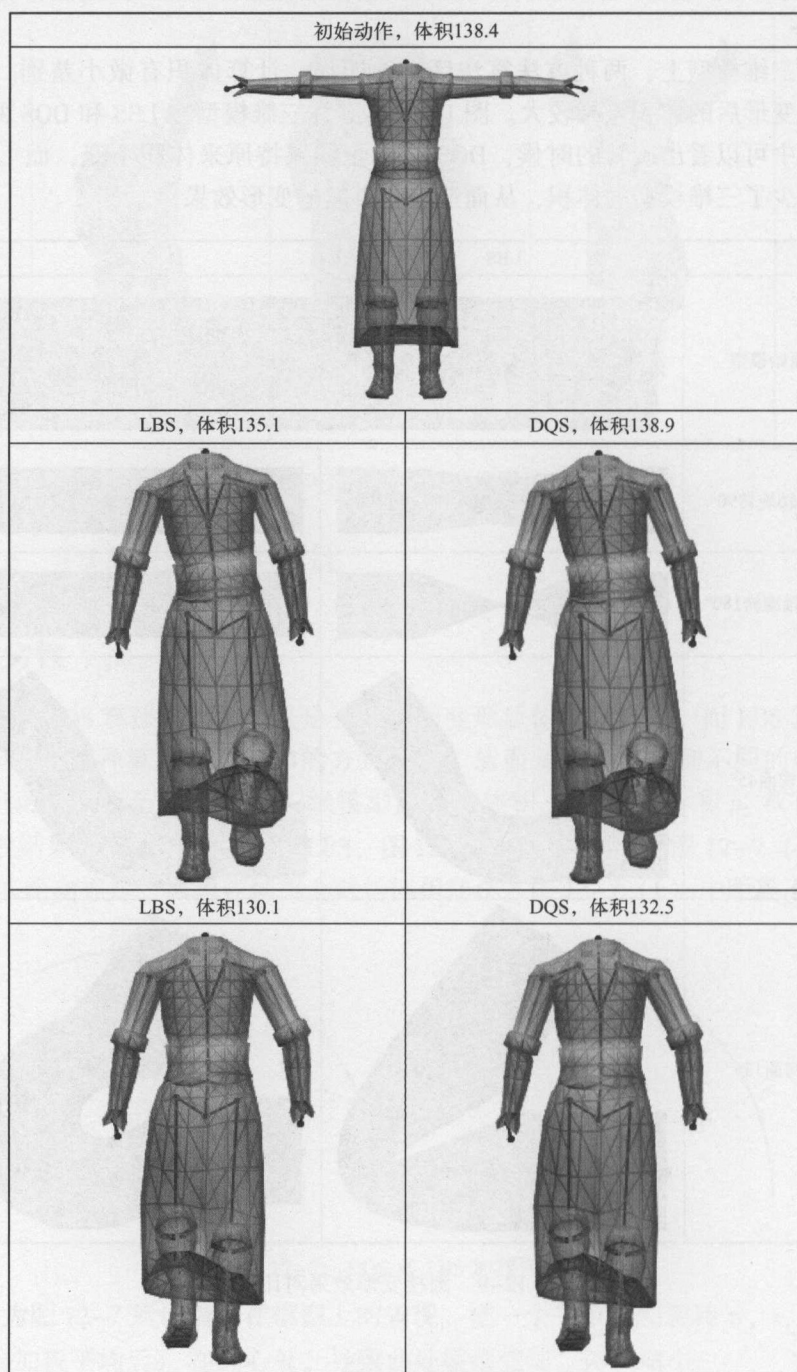


图 12-10 两种算法变形后体积对比

是三维模型形状会发生变化。在图 12-11 中，只旋转最右边末端关节点得到的三维模型两种算法变形的效果不一样，但是骨骼的整体形都保持不变。

图 12-12 是具有 5 个关节点的多段三维模型进行变形的效果，从中可以看出 DQS 仍然能够更好地保持体积不变。

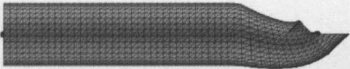

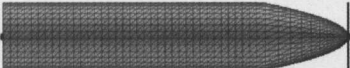

算法	LBS	DQS
末端节点 旋转90°		
末端节点 旋转180°		

图 12-11 末端节点旋转效果

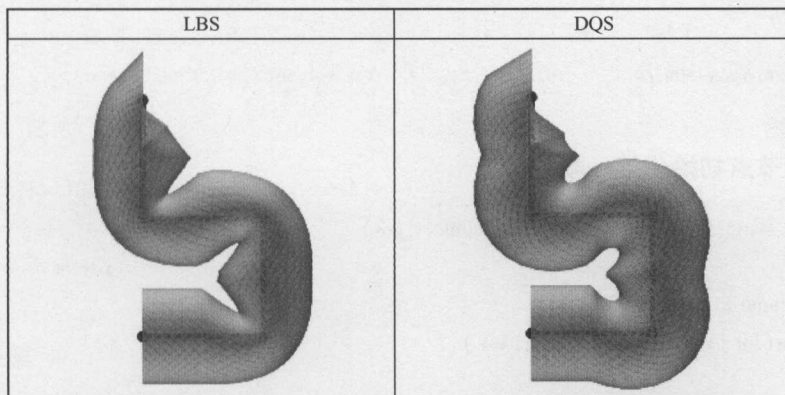


图 12-12 多段三维模型末端节点旋转效果

12.5.2 测试模型生成

蒙皮数据可以从文件中加载，也可以通过算法生成。上面使用的圆柱三维模型由于形状比较简单，因此可以通过算法生成关节的位置，以及相应的蒙皮权重数据。整个算法一共分为 6 个函数，分别是生成关节位置、生成骨骼层次、生成关节初始化第一帧数据、生成三维模型、生成权重、生成完整的蒙皮数据。

1. 生成关节位置

```
private static double[] CreateJointPos(int jointNum)
{
    double[] arr = new double[jointNum];
    for(int i = 0; i < jointNum; i++)
    {
        arr[i] = -0.5 + i / (jointNum - 1d);
    }
    return arr;
}
```

2. 生成骨骼层次

```
private static JointHierarchy[] CreateSkeleton(int jointNum)
{
}
```

```

JointHierarchy[ ] skeleton = new JointHierarchy[ jointNum ];
for( int i = 0; i < jointNum; i ++ )
{
    skeleton[ i ] = new JointHierarchy( )
    {
        Index = i,
        Name = i. ToString( ),
        Parent = i - 1
    };
}
return skeleton;
}

```

3. 生成关节点初始化第一帧数据

```

private static Animation CreatePose( double[ ] pos )
{
    Frame f0 = new Frame( );
    for( int i = 0; i < pos. Length; i ++ )
    {
        double length = i == 0 ? pos[ i ] : pos[ i ] - pos[ i - 1 ];
        f0. Add( new JointFrame( ) { Position = new Vector3D( length, 0, 0 ) } );
    }
    Animation ani = new Animation( );
    ani. Add( f0 );
    return ani;
}

```

4. 生成圆柱三维模型

```

private static TriMesh CreateMesh( double[ ] pos, double r, double density )
{
    TriMesh mesh = new TriMesh( );
    int m = ( int ) ( Math. Abs( pos[ 0 ] - pos[ pos. Length - 1 ] ) * density );
    int n = ( int ) ( r * Math. PI * 2 * density );
    double t = Math. PI * 2 / n;
    TriMesh. Vertex[ , ] arr = new HalfEdgeMesh. Vertex[ m, n ];
    for( int i = 0; i < m; i ++ )
    {
        double x = pos[ 0 ] + i / ( m - 1 ) * d;
        for( int j = 0; j < n; j ++ )
        {
            double y = r * Math. Cos( t * j );
            double z = r * Math. Sin( t * j );

```

```

        arr[i,j] = mesh.Vertices.Add(new VertexTraits(x,y,z));
    }
}

for(int i=1;i<m;i++)
{
    for(int j=0;j<n;j++)
    {
        int k=(j+1)%n;
        mesh.Faces.AddTriangles(arr[i,j],arr[i-1,j],arr[i-1,k]);
        mesh.Faces.AddTriangles(arr[i,k],arr[i,j],arr[i-1,k]);
    }
}

TriMeshUtil.SetUpNormalVertex(mesh);

return mesh;
}

```

5. 生成权重

```

private static WeightPair[ ][ ] CreateWeight( double[ ] pos, TriMesh mesh)
{
    //权重
    //pos0          pos1          pos2
    //      mid0          mid1      mid2
    // | ---- : ---- | ---- : ---- |
    // |  w0   |      w01   | w12   |
    WeightPair[ ][ ] weights = new WeightPair[ mesh.Vertices.Count ][ ];
    double[ ] mid = new double[ pos.Length ];
    mid[ pos.Length - 1 ] = pos[ pos.Length - 1 ];
    for(int i=0;i<pos.Length-1;i++)
    {
        mid[i] = ( pos[i] + pos[i+1] )/2;
    }
    for(int i=0;i<mesh.Vertices.Count;i++)
    {
        double x = mesh.Vertices[i].Traits.Position.x;
        int left = pos.Length - 1;
        int right = left;
        for(int j=0;j<pos.Length;j++)
        {
            if(x < mid[j])

```



```

        {
            left = j == 0? j:j - 1;
            right = j;

            break;
        }
    }
    if(left != right)
    {
        weights[i] = new WeightPair[2];
        double weight = x < pos[right]?
            (x - mid[left]) / (pos[right] - mid[left]) * 0.5;
            (x - pos[right]) / (mid[right] - pos[right]) * 0.5 + 0.5;
        weights[i][0] = new WeightPair() { Joint = left, Weight = 1 - weight };
        weights[i][1] = new WeightPair() { Joint = right, Weight = weight };
    }
    else
    {
        weights[i] = new WeightPair[1];
        weights[i][0] = new WeightPair() { Joint = right, Weight = 1 };
    }
}

return weights;
}

```

6. 生成完整蒙皮数据

```

public static SMDSysytem CreateCylinder(int jointNum, double r, double density)
{
    double[] pos = CreateJointPos(jointNum);
    JointHierarchy[] skeleton = CreateSkeleton(jointNum);
    Animation ani = CreatePose(pos);
    TriMesh mesh = CreateMesh(pos, r, density);
    WeightPair[][] weight = CreateWeight(pos, mesh);

    List<Animation> animations = new List<Animation>();
    animations.Add(ani);
    SMDSysytem smdSystem = new SMDSysytem(mesh, weight, skeleton, animations);
    return smdSystem;
}

```

7. 图示

图 12-13 是三个关节、五个关节、十个关节的圆柱蒙皮三维模型。

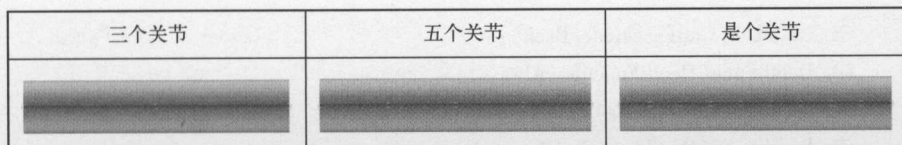


图 12-13 不同数量关节圆柱三维模型图示



12.6 蒙皮显示

在渲染三维骨骼动画的时候，常常需要显示三维模型的骨骼，并把三维模型显示为半透明效果，这样就可以同时看到三维模型和骨骼，从而可以更加真实地表现骨骼的变动驱动了三维模型的变形。蒙皮的显示系统函数如下所示，主要分为设置透明状态、骨骼绘制、三维模型绘制三大部分。

1. 设置透明状态

```
private void DrawBlend()
{
    GlobalSetting.SettingLight.DoubleSideLight = false;
    GL.Enable(EnableCap.ColorMaterial);
    GL.Enable(EnableCap.CullFace);
    GL.CullFace(CullFaceMode.Back);
    GL.Enable(EnableCap.DepthTest);
    GL.Enable(EnableCap.Blend);
    GL.BlendFunc(BlendingFactorSrc.SrcAlpha,
        BlendingFactorDest.OneMinusSrcAlpha);

    this.DrawSkeleton();
    GL.Disable(EnableCap.ColorMaterial);
    GL.DepthMask(false);

    float f = 1 - GlobalSetting.MaterialSetting.MaterialDiffuse.A/256f;
    float alpha = GlobalSetting.MaterialSetting.MaterialDiffuse.A/256f;

    GL.Enable(EnableCap.CullFace);
    GL.CullFace(CullFaceMode.Front);
    GL.DepthFunc(DepthFunction.Always);
    this.DrawModel(f * alpha);
    GL.DepthFunc(DepthFunction.Lequal);
    this.DrawModel((alpha - f * alpha) / (1f - f * alpha));
    this.DrawSkeleton();

    GL.Enable(EnableCap.CullFace);
```

```

GL. CullFace( CullFaceMode. Back );
GL. DepthFunc( DepthFunction. Always );
this. DrawModel( f * alpha );
GL. DepthFunc( DepthFunction. Lequal );
this. DrawModel( ( alpha - f * alpha ) / ( 1f - f * alpha ) );

GL. Disable( EnableCap. Blend );
GL. DepthMask( true );
GL. Disable( EnableCap. DepthTest );
}

```

2. 三维模型绘制

```

private void DrawModel( float alpha )
{
    OpenTK. Graphics. Color4 front =
        GlobalSetting. MaterialSetting. MaterialDiffuse;
    front. A = alpha;
    OpenTK. Graphics. Color4 back =
        GlobalSetting. MaterialSetting. BackMaterialDiffuse;
    back. A = alpha;
    GL. Material( MaterialFace. Front, MaterialParameter. Diffuse, front );
    GL. Material( MaterialFace. Back, MaterialParameter. Diffuse, back );
    this. DrawModel();
}

private void DrawModel()
{
    TriMesh mesh = SMDGlobal. Instance. smdSystem. MeshCurrent;
    GL. Enable( EnableCap. PolygonOffsetFill );
    GL. ShadeModel( ShadingModel. Smooth );
    GL. PolygonMode( GlobalSetting. EnalbeCapsSetting. PolygonFace,
        PolygonMode. Fill );
    GL. Enable( EnableCap. Normalize );
    Color c = GlobalSetting. DisplaySetting. MeshColor;
    OpenGLManager. Instance. SetColor( c );
    DrawTriangles( mesh );

    GL. PolygonMode( GlobalSetting. EnalbeCapsSetting. PolygonFace,
        PolygonMode. Line );
    GL. LineWidth( GlobalSetting. DisplaySetting. LineWidth );
    GL. Enable( EnableCap. CullFace );
    OpenGLManager. Instance. SetColor(
        GlobalSetting. DisplaySetting. WifeFrameColor );
}

```



```

    DrawTriangles( mesh );
    GL. Disable( EnableCap. PolygonOffsetFill );
}
private void DrawTriangles( TriMesh mesh )
{
    GL. Begin( BeginMode. Triangles );
    for( int i = 0; i < mesh. Faces. Count; i ++ )
    {
        foreach( TriMesh. Vertex vertex in mesh. Faces[ i ]. Vertices )
        {
            GL. Normal3( vertex. Traits. Normal. ToArray( ) );
            GL. Vertex3( ( ( vertex. Traits. Position - this. offset )
                / this. scale ). ToArray( ) );
        }
    }
    GL. End( );
}

```

3. 骨骼绘制

```

private void DrawSkeleton( )
{
    SMDSystem model = SMDGlobal. Instance. smdSystem;
    GL. Enable( EnableCap. PointSmooth );
    GL. Enable( EnableCap. LineSmooth );
    GL. Hint( HintTarget. PointSmoothHint, HintMode. Fastest );
    GL. Hint( HintTarget. LineSmoothHint, HintMode. Fastest );
    GL. Enable( EnableCap. ColorMaterial );
    GL. Color3( Color. FromArgb( SMDConfig. Instance. JointColor ) );
    GL. PointSize( ( float ) SMDConfig. Instance. PointSize );
    GL. Begin( BeginMode. Points );
    for( int i = 0; i < model. joints. Length; i ++ )
    {
        GL. Vertex3( ( ( model. GetJoint( i ) - this. offset )
            / this. scale ). ToArray( ) );
    }
    GL. End( );

    GL. Color3( Color. FromArgb( SMDConfig. Instance. BoneColor ) );
    GL. LineWidth( ( float ) SMDConfig. Instance. LineWidth );
    GL. Begin( BeginMode. Lines );
    for( int i = 1; i < model. joints. Length; i ++ )
    {

```

```
Vector3D a = model.GetJoint(i);  
Vector3D b = model.GetJointParent(i);  
GL.Vertex3(((model.GetJoint(i) - this.offset)  
    /this.scale).ToArray());  
GL.Vertex3(((model.GetJointParent(i) - this.offset)  
    /this.scale).ToArray());  
}  
GL.End();  
GL.Disable(EnableCap.ColorMaterial);  
}
```

第 13 章

曲线



13.1 参数化曲线

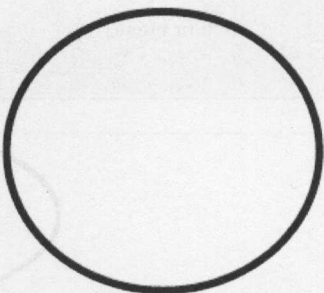
三维空间里的曲线有多种表示方法。例如，参数化表示方法、隐函数表示方法。参数化曲线指的是曲线上点的位置表示为一个变量的函数，即曲线上点的坐标 x 、 y 、 z 分别是此变量的函数。坐标也可以用极坐标等其他坐标形式来表示。曲线是一维图形，因此只需要一个变量。例如，对于图 13-1 的圆形来说，参数化表示方法为

$$f(t) = \begin{pmatrix} r\cos t \\ r\sin t \end{pmatrix}, S = f([0, 2\pi])$$

隐函数表示方法为

$$f(x,y) = x^2 + y^2 - r^2$$
$$S = \{(x,y) \in R^2 | f(x,y) = 0\}$$

图 13-1 圆形曲线



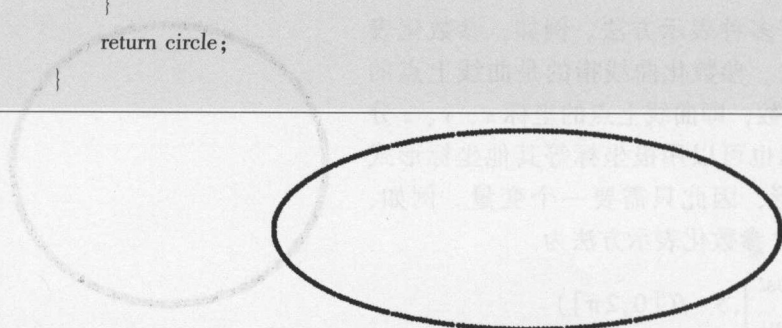
很多曲线可以用公式来表示，这些曲线称为简单曲线，如下面的一些曲线。

直 线

```
public TriMesh CreateMeshLine()  
{  
    TriMesh line = new TriMesh();  
    for(int t = 0; t < VerticesNum; t++)  
    {  
        line.Vertices.Add(new VertexTraits(-1 + t * 0.001,  
                                             (-1 + t * 0.001) / Math.Tan(20),  
                                             0));  
    }  
    return line;  
}
```

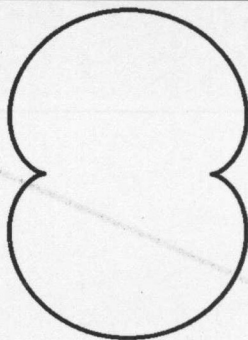

椭圆

```
public TriMesh CreateMeshCricle(double radius)
{
    TriMesh circle = new TriMesh();
    for(int t=0;t<VerticesNum;t++)
    {
        circle.Vertices.Add(new VertexTraits(0.2 * radius * Math.Cos(360 * t/VerticesNum),
        0.5 * radius * Math.Sin(360 * t/VerticesNum),
        0));
    }
    return circle;
}
```



双圆弧线

```
public TriMesh CreateDoubleArcEpicycloidLine(double l,double b)
{
    TriMesh dael = new TriMesh();
    for(int t=0;t<VerticesNum;t++)
    {
        dael.Vertices.Add(new VertexTraits(3 * b * Math.Cos(t * 360) + l * Math.Cos(3 * t * 360),
        3 * b * Math.Sin(t * 360) + l * Math.Sin(3 * t * 360),
        0));
    }
    return dael;
}
```

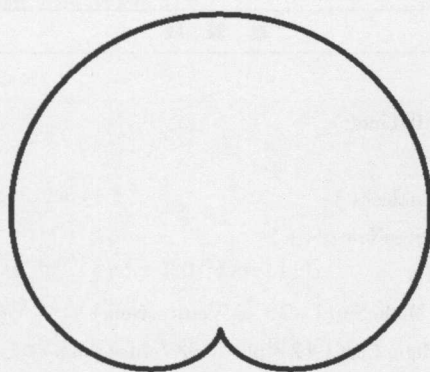


心 形 线

```

public TriMesh CreateHeartLine( double a)
{
    TriMesh heart = new TriMesh();
    for( int t = 0; t < VerticesNum; t++)
    {
        heart.Vertices.Add( new VertexTraits( a * ( 1 + Math. Cos( 360 * t ) ) * Math. Cos( 360 * t ) ,
                                                a * ( 1 + Math. Cos( 360 * t ) ) * Math. Sin( 360 * t ) ,
                                                0 ) );
    }
    return heart;
}

```

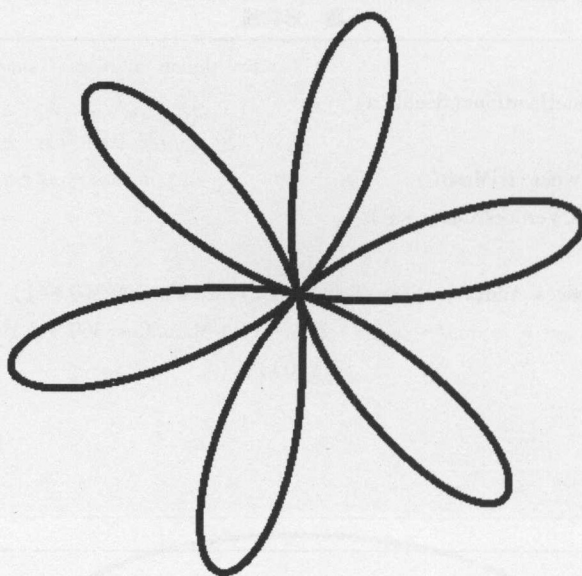


花 曲 线

```

public TriMesh CreateCircularHelixLine()
{
    TriMesh chl = new TriMesh();
    for( int t = 0; t < VerticesNum; t++)
    {
        chl.Vertices.Add( new VertexTraits ( ( 10 + 10 * Math. Sin( 6 * t * 360 ) ) * Math. Cos( t * 360 ) ,
                                                ( 10 + 10 * Math. Sin( 6 * t * 360 ) ) * Math. Sin( t * 360 ) ,
                                                2 * Math. Sin( 6 * t * 360 ) ) );
    }
    return chl;
}

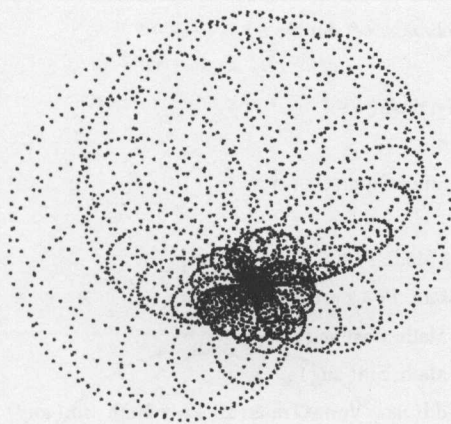
```



蝴蝶线

```

public TriMesh CreateButterflyLine()
{
    TriMesh circle = new TriMesh();
    for(int i=0;i <= VerticesNum;i++)
    {
        double x = ((4 * Math. Sin(i * 2 * pi/VerticesNum) +
                    6 * Math. Cos(i * 2 * pi * 360/VerticesNum))
                    * Math. Sin(i * 2 * pi/VerticesNum)
                    * Math. Cos(Math. Log(1 + i * 2 * pi/VerticesNum)
                    * i * 2 * pi/VerticesNum));
        double y = ((4 * Math. Sin(i * 2 * pi/VerticesNum) +
                    6 * Math. Cos(i * 2 * pi * 360/VerticesNum))
                    * Math. Sin(i * 2 * pi/VerticesNum) *
                    Math. Sin(Math. Log(1 + i * 2 * pi/VerticesNum)
                    * i * 2 * pi/VerticesNum));
        double z = ((4 * Math. Sin(i * 2 * pi/VerticesNum) +
                    6 * Math. Cos(i * 2 * pi * 360/VerticesNum))
                    * Math. Cos(i * 2 * pi/VerticesNum));
        circle.Vertices.Add(new VertexTraits(x,y,z));
    }
    return circle;
}
    
```

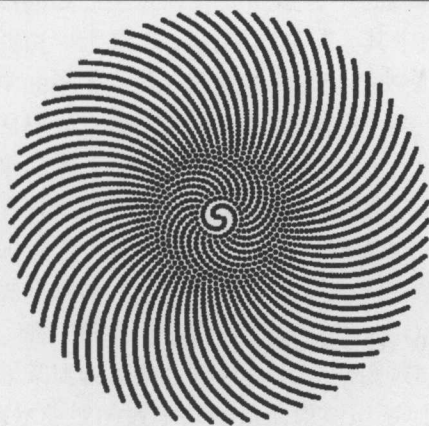



费马线

```

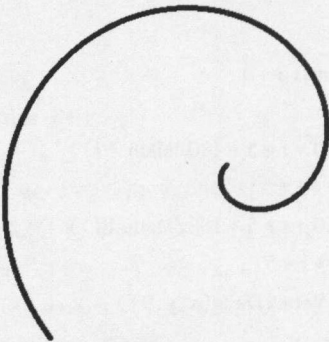
public TriMesh CreateFermatCurve(double a)
{
    TriMesh fc = new TriMesh();
    double x = 0;
    double y = 0;
    for (int t = 0; t < VerticesNum; t++)
    {
        x = (a * Math.Sqrt(360 * t * 5 * 180 / Math.PI))
            * Math.Cos(360 * t * 5);
        y = (a * Math.Sqrt(360 * t * 5 * 180 / Math.PI))
            * Math.Sin(360 * t * 5);
        fc.Vertices.Add(new VertexTraits(x, y, 0));
    }
    return fc;
}

```



渐开线

```
public TriMesh CreateInvoluteCurve(double a)
{
    TriMesh line = new TriMesh();
    int r = 1;
    for(int t = 0; t <= VerticesNum; t++)
    {
        double ang = 2 * Math.PI * t / VerticesNum;
        double s = 2 * Math.PI * r * t / VerticesNum;
        double x0 = s * Math.Cos(ang);
        double y0 = s * Math.Sin(ang);
        line.Vertices.Add(new VertexTraits(x0 + s * Math.Sin(ang),
                                            y0 - s * Math.Cos(ang),
                                            0));
    }
    return line;
}
```



外旋轮线

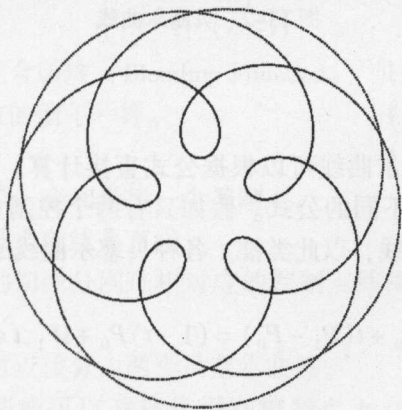
```
public TriMesh CreateEpitrochoid()
{
    TriMesh line = new TriMesh();
    float a = 0.5F;
    float b = 0.3F;
    float c = 0.5F;
    float theta;
    for(int i = 0; i < 20000; i++)
    {
        theta = i * 360 * 10 / VerticesNum;
```

```

line.Vertices.Add(new VertexTraits(((a + b) * Math.Cos(theta)
    - c * Math.Cos((a/b + 1) * theta))/10,
    ((a + b) * Math.Sin(theta) -
    c * Math.Sin((a/b + 1) * theta))/10,
    0));
}

return line;
}

```



13.2 贝塞尔曲线

13.2.1 概述

贝塞尔曲线 (Bézier curve) 是三维图形中最重要的一种数学曲线。各种矢量图形软件使用贝塞尔曲线精确制作曲线。贝塞尔曲线由线段与控制点组成, 控制点是可拖动的支点, 线段像可伸缩的皮筋。贝塞尔曲线在生成时首先确定各个控制点, 根据控制点的路径和描绘的先后顺序再生成曲线。曲线由一个或多个曲线段组成。其中的控制点标记路径段的端点, 移动这些控制点将改变路径中曲线的形状。贝塞尔曲线 (Bézier Curve) 是由控制点生成的参数化的矢量曲线, 扩展到曲面称为贝塞尔曲面。在显示的时候, 可以在空间上计算曲线上点的位置, 并把点连接成线段来显示。

一个贝塞尔曲线由 $P_0 \sim P_n$ 个控制点生成, n 是它的度数, 当 $n=1$, 即两个控制点的时候, 贝塞尔曲线是线性的 (Linear); 当 $n=2$, 也就是三个控制点, 曲线是二次的 (Quadratic); 以此类推, 有三次、四次等贝塞尔曲线。第一个和最后一个控制点是曲线的两个端点, 通常中间的控制点不落在曲线上。通用的贝塞尔曲线度数 (Degree) 是二次 (Quadratic) 和三次 (Cubic) 的。更高度数计算起来比较复杂, 通常用几个低度数的曲线连接起来构成更复杂的曲线 (Bezier-Spline)。贝塞尔曲线的数学基础是伯恩斯坦多项式 (Bernstein Basis Polynomi-

als)。例如，图 13-2 是 3 个控制点生成的贝塞尔曲线。其中，蓝色的线是贝塞尔曲线，红色的线是控制点的连线。

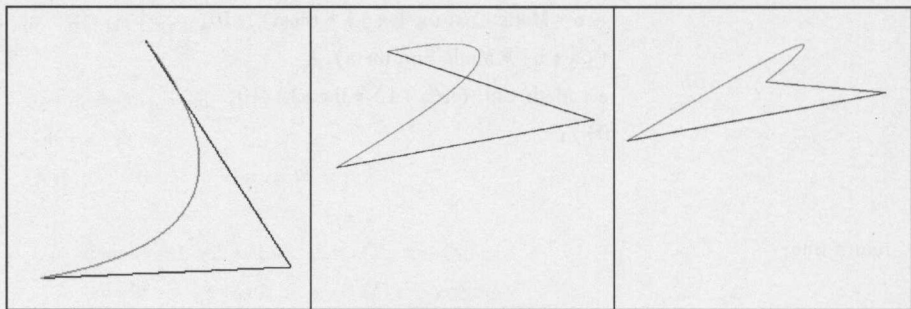


图 13-2 贝塞尔曲线

13.2.2 贝塞尔曲线公式

在控制点确定以后，贝塞尔曲线可以根据公式直接计算。贝塞尔曲线的公式不是唯一的，根据控制点数量的不同有不同的公式。假如只有两个控制点，则称为线性贝塞尔曲线。三个控制点称为二次贝塞尔曲线，以此类推。各种贝塞尔曲线的公式如下。

1. 线性贝塞尔曲线公式

$$B(t) = P_0 + t(P_1 - P_0) = (1-t)P_0 + tP_1, t \in [0, 1]$$

2. 二次贝塞尔曲线公式

$$B(t) = (1-t)[(1-t)P_0 + tP_1] + t[(1-t)P_1 + tP_2], t \in [0, 1]$$

二次曲线可以看作 P_0 到 P_1 和 P_1 到 P_2 两个一次曲线的连接，上述公式重写为

$$B(t) = (1-t)^2P_0 + 2(1-t)tP_1 + t^2P_2, t \in [0, 1]$$

曲线相对 t 的导数是

$$B'(t) = 2(1-t)(P_1 - P_0) + 2t(P_2 - P_1), t \in [0, 1]$$

从中可以看出第一段曲线的切线和第二段曲线的切线在 P_1 处一致。二次贝塞尔曲线也是一个抛物线 (Parabolic) 段，也就是圆锥 (Conic Section) 曲线。

3. 三次贝塞尔曲线公式：

四个控制点决定一个三次贝塞尔曲线，通常 P_1 和 P_2 不和曲线相交，只控制曲线的方向，某些 P_1 和 P_2 可以造成曲线自我相交，或者形成尖端 (Cusp)。

$$B(t) = (1-t)^3P_0 + 3(1-t)^2tP_1 + 3(1-t)t^2P_2 + t^3P_3, t \in [0, 1]$$

也可写成两个二次贝塞尔曲线的和

$$B(t) = (1-t)B_{P_0, P_1, P_2}(t) + tB_{P_1, P_2, P_3}(t), t \in [0, 1]$$

4. N 次贝塞尔曲线公式

N 次贝塞尔曲线可以递归定义，也就是表示为两个 $N-1$ 次贝塞尔曲线之和，即

$$B(t) = B_{P_0 P_1 \dots P_n}(t) = (1-t)B_{P_0 P_1 \dots P_{n-1}}(t) + tB_{P_1 P_2 \dots P_n}(t)$$

不用递归，此公式显示定义如下：

$$B(t) = \sum_{i=0}^n \binom{n}{i} (1-t)^{n-i} t^i P_i$$

$$= (1-t)^n P_0 + \binom{n}{1} (1-t)^{n-1} t P_1 + \cdots + \binom{n}{n-1} (1-t) t^{n-1} P_{n-1} + t^n P_n, t \in [0, 1]$$

例如, 5次贝塞尔曲线公式为

$$B_{P_0 P_1 P_2 P_3 P_4 P_5}(t) = B(t) = (1-t)^5 P_0 + 5t(1-t)^4 P_1 + 10t^2(1-t)^3 P_2 + 10t^3(1-t)^2 P_3 + 5t^4(1-t) P_4 + t^5 P_5, t \in [0, 1]$$

下面的多项式称为度数为 n 的伯恩斯坦多项式:

$$b_{i,n}(t) = \binom{n}{i} t^i (1-t)^{n-i}, i=0, \cdots, n$$

其中, 二项式系数 (Binomial Coefficient) 为

$$\binom{n}{i} = \frac{n!}{i! (n-i)!}$$

伯恩斯坦多项式是控制点的混合函数 (Blending Function), 贝塞尔曲线上的每个点受到所有控制点的影响, 只是混合函数的值不一样。

5. 贝塞尔曲线的特点

- (1) 曲线的两个端点是第一个和最后一个控制点。
- (2) 如果控制点共线, 那么曲线是直线。
- (3) 曲线在两个端点处的切线分别是相对应的控制点线段, 也就是切线分别是 $P_0 P_1$ 和 $P_{n-1} P_n$ 。
- (4) 一条曲线可以在任意点被分为两个贝塞尔曲线。
- (5) 旋转、位移等放射变换可以直接作用在控制点上, 得到和作用于曲线上一样的效果。
- (6) 圆等形状无法用贝塞尔曲线表示; 尽管可以用四个三次贝塞尔曲线近似于一个圆, 但是仍旧有误差。
- (7) 和贝塞尔曲线平行的、有固定位移的曲线无法用贝塞尔曲线进行构造。
- (8) 控制点生成的贝塞尔曲线落在控制点的凸多边形外包框 (Convex Hull) 内部。
- (9) 每个二次曲线也是一个三次曲线, 每个 n 次曲线也是一个 m 次 ($n < m$) 曲线, 即 P_0, \cdots, P_n 个点的 n 次曲线相等于 P'_0, \cdots, P'_{n+1} 个点的 $n+1$ 次曲线, 其中控制点的关系是

$$P'_k = \frac{k}{n+1} P_{k-1} + \left(1 - \frac{k}{n+1}\right) P_k$$

13.2.3 度数升级

一个 n 度数的贝塞尔曲线可以变为一个度数为 $n+1$ 的同样形状的贝塞尔曲线。度数升级指的是把贝塞尔曲线从度数为 n 变为度数为 $n+1$, 但是曲线的形状不变, 还是原来的曲线。度数升级的过程如下。

第一步: 根据 $B(t) = (1-t)B(t) + tB(t)$, 然后公式里面的每个项 $b_{i,n}(t)P_i$ 被乘以 $(1-t)$ 或 t , 就可以得到更高一级的曲线。

第二步: 例如, 从二次到三次的升级如下。

$$\begin{aligned}
 & (1-t)^2 P_0 + 2(1-t)tP_1 + t^2 P_2 \\
 &= (1-t)^3 P_0 + (1-t)^2 tP_0 + 2(1-t)^2 tP_1 \\
 & \quad + 2(1-t)t^2 P_1 + (1-t)t^2 P_2 + t^3 P_2 \\
 &= (1-t)^3 P_0 + 3(1-t)^2 t \frac{P_0 + 2P_1}{3} + 3(1-t)t^2 \frac{2P_1 + P_2}{3} + t^3 P_2
 \end{aligned}$$

第三步：对于任意度数 n ，升级公式如下：

$$\binom{n+1}{i}(1-t)b_{i,n} = \binom{n}{i}b_{i,n+1}, (1-t)b_{i,n} = \frac{n+1-i}{n+1}b_{i,n+1}$$

$$\binom{n+1}{i}(1-t)b_{i,n} = \binom{n}{i}b_{i,n+1}, (1-t)b_{i,n} = \frac{n+1-i}{n+1}b_{i,n+1}$$

$$\begin{aligned}
 B(t) &= (1-t) \sum_{i=0}^n b_{i,n}(t)P_i + t \sum_{i=0}^n b_{i,n}(t)P_i \\
 &= \sum_{i=0}^n \frac{n+1-i}{n+1} b_{i,n+1}(t)P_i + \sum_{i=0}^n \frac{i+1}{n+1} b_{i+1,n+1}(t)P_i \\
 &= \sum_{i=0}^{n+1} \left(\frac{i}{n+1} P_{i-1} + \frac{n+1-i}{n+1} P_i \right) b_{i,n+1}(t) \\
 &= \sum_{i=0}^{n+1} b_{i,n+1}(t)P'_i
 \end{aligned}$$

第四步：新的控制点由旧的控制点生成，公式如下：

$$P'_i = \frac{i}{n+1}P_{i-1} + \frac{n+1-i}{n+1}P_i, i=0, \dots, n+1$$

1. 贝塞尔曲线的多项式表示形式

把伯恩斯坦多项式形式化简为直接的多项式形式，贝塞尔曲线的公式为

$$B(t) = \sum_{j=0}^n t^j C_j$$

$$\text{其中, } C_j = \frac{n!}{(n-j)!} \sum_{i=0}^j \frac{(-1)^{i+j} P_i}{i!(j-i)!} = \prod_{m=0}^{j-1} (n-m) \sum_{i=0}^j \frac{(-1)^{i+j} P_i}{i!(j-i)!}$$

对于高度数的曲线，这种表示形式在数值计算时可能会出现数值计算稳定性问题。因此需要用德卡斯特里奥（De Casteljau's Algorithm）算法来计算。

2. 德卡斯特里奥算法

德卡斯特里奥（De Casteljau's Algorithm）算法是一种计算伯恩斯坦多项式递归的方法，可以用来把一个贝塞尔曲线在某个任意参数处分割为两个贝塞尔曲线，这种算法具有高度的数值计算稳定性。

第一步：贝塞尔曲线计算公式为

$$B(t) = \sum_{i=0}^n \beta_i b_{i,n}(t)$$

第二步：伯恩斯坦多项式为

$$b_{i,n}(t) = \binom{n}{i}(1-t)^{n-i}t^i$$

第三步：从 n 个控制点可以生成新的控制点。

$$\beta_i^{(0)} := \beta_i, i = 0, \dots, n$$

$$\beta_i^{(j)} := \beta_i^{(j-1)}(1 - t_0) + \beta_{i+1}^{(j-1)}t_0, i = 0, \dots, n-j, j = 1, \dots, n$$

第四步：最终贝塞尔曲线在 t_0 点的值为

$$B(t_0) = \beta_0^{(n)}$$

第五步：从中可以看出贝塞尔曲线可以在点 t_0 分割为两个曲线，每个曲线的控制点分别为

$$\begin{aligned} &\beta_0^{(0)}, \beta_0^{(1)}, \dots, \beta_0^{(n)} \\ &\beta_0^{(n)}, \beta_1^{(n-1)}, \dots, \beta_n^{(0)} \end{aligned}$$

3. 德卡斯特里奥算法的几何解释

(1) 把控制点连接成线段。

(2) 把每个线段以 $t:(1-t)$ 的比列分割，把新的控制点连接起来，此时控制点比上一级少一个。

(3) 重复上两步，直到只有一个控制点，此控制点的位置就是贝塞尔曲线的值。

13.2.4 贝塞尔曲线代码

贝塞尔曲线的代码很简单，可以根据公式直接进行计算。其中函数的输入是控制点的位置，输出一组位于贝塞尔曲线上的点。贝塞尔曲线的代码根据曲线度数的不同而不同，每种度数的贝塞尔曲线生成需要两步，第一步生成控制点，第二步生成曲线。第一步生成的控制点的位置是随机的，可以进行手工调整。

1. 二次贝塞尔曲线代码

1) 生成控制点

```
public TriMesh CreateThreeBezierControlPoint()
{
    TriMesh controlPoint = new TriMesh();
    controlPoint.Vertices.Add(new VertexTraits(0,0,0));
    controlPoint.Vertices.Add(new VertexTraits(0.2,0.2,0));
    controlPoint.Vertices.Add(new VertexTraits(0.4,0.5,0));
    return controlPoint;
}
```

2) 生成贝塞尔曲线

```
public TriMesh CreateThreeBezierCurve(TriMesh mesh)
{
    TriMesh curve = new TriMesh();
    double x = 0;
    double y = 0;
    double z = 0;
    for(int t = 0; t < VerticesNum; t++)
    {

```

```

double tt = (double)t / (double)VerticesNum;
x = mesh.Vertices[0].Traits.Position.x * Math.Pow(1 - tt, 2)
    + 2 * mesh.Vertices[1].Traits.Position.x * tt * (1 - tt)
    + mesh.Vertices[2].Traits.Position.x * Math.Pow(tt, 2);
y = mesh.Vertices[0].Traits.Position.y * Math.Pow(1 - tt, 2)
    + 2 * mesh.Vertices[1].Traits.Position.y * tt * (1 - tt)
    + mesh.Vertices[2].Traits.Position.y * Math.Pow(tt, 2);
z = mesh.Vertices[0].Traits.Position.z * Math.Pow(1 - tt, 2)
    + 2 * mesh.Vertices[1].Traits.Position.z * tt * (1 - tt)
    + mesh.Vertices[2].Traits.Position.z * Math.Pow(tt, 2);
curve.Vertices.Add(new VertexTraits(x, y, z));
}
return curve;
}

```

2. 三次贝塞尔曲线代码

1) 生成控制点

```

public TriMesh CreateFourBezierControlPoint()
{
    TriMesh controlPoint = new TriMesh();
    controlPoint.Vertices.Add(new VertexTraits(0, 0, 0));
    controlPoint.Vertices.Add(new VertexTraits(0.2, 0.2, 0));
    controlPoint.Vertices.Add(new VertexTraits(0.4, 0.5, 0));
    controlPoint.Vertices.Add(new VertexTraits(0.6, 0.4, 0));
    return controlPoint;
}

```

2) 生成贝塞尔曲线

```

public TriMesh CreateFourBezierCurve(TriMesh mesh)
{
    TriMesh curve = new TriMesh();
    double x = 0;
    double y = 0;
    double z = 0;
    for (int t = 0; t < VerticesNum; t++)
    {
        double tt = (double)t / (double)VerticesNum;
        x = mesh.Vertices[0].Traits.Position.x * Math.Pow(1 - tt, 3)
            + 3 * mesh.Vertices[1].Traits.Position.x * tt * Math.Pow(1 - tt, 2)
            + 3 * mesh.Vertices[2].Traits.Position.x * Math.Pow(tt, 2) * (1 - tt)
            + mesh.Vertices[3].Traits.Position.x * Math.Pow(tt, 3);

```

```

y = mesh.Vertices[0].Traits.Position.y * Math.Pow(1 - tt, 3)
    + 3 * mesh.Vertices[1].Traits.Position.y * tt * Math.Pow(1 - tt, 2)
    + 3 * mesh.Vertices[2].Traits.Position.y * Math.Pow(tt, 2) * (1 - tt)
    + mesh.Vertices[3].Traits.Position.y * Math.Pow(tt, 3);
z = mesh.Vertices[0].Traits.Position.z * Math.Pow(1 - tt, 3)
    + 3 * mesh.Vertices[1].Traits.Position.z * tt * Math.Pow(1 - tt, 2)
    + 3 * mesh.Vertices[2].Traits.Position.z * Math.Pow(tt, 2) * (1 - tt)
    + mesh.Vertices[3].Traits.Position.z * Math.Pow(tt, 3);
curve.Vertices.Add(new VertexTraits(x, y, z));
}
return curve;
}

```

3. N次贝塞尔曲线代码

1) 生成控制点

```

public TriMesh CreateNBezierControlPoint()
{
    TriMesh cPoint = new TriMesh();
    Random r = new Random();
    cPoint.Vertices.Add(new VertexTraits(0, 0, 0));
    for(int t = 0; t < controlPoint; t++)
    {
        int x = r.Next(1, 10);
        cPoint.Vertices.Add(new VertexTraits(t * 0.1 + x * 0.01,
                                                x * 0.1,
                                                0));
    }
    return cPoint;
}

```

2) 生成曲线

```

public TriMesh CreateNBezierCurve(TriMesh mesh)
{
    TriMesh NBezierCurve = new TriMesh();
    double x = 0;
    double y = 0;
    double z = 0;
    for(int t = 0; t < VerticesNum; t++)
    {
        double tt = (double)t / (double)VerticesNum;
        for(int i = 0; i < = controlPoint; i++)
        {

```



```

        x = mesh.Vertices[i].Traits.Position.x * XX(controlPoint,i)
            * Math.Pow(1 - tt,controlPoint - i) * Math.Pow(tt,i) + x;
        y = mesh.Vertices[i].Traits.Position.y * XX(controlPoint,i)
            * Math.Pow(1 - tt,controlPoint - i) * Math.Pow(tt,i) + y;
        z = mesh.Vertices[i].Traits.Position.z * XX(controlPoint,i)
            * Math.Pow(1 - tt,controlPoint - i) * Math.Pow(tt,i) + z;
    }
    NBezierCurve.Vertices.Add(new VertexTraits(x,y,z));
    x=0;
    y=0;
    z=0;
}

return NBezierCurve;
}

private int XX(int n,int a)
{
    int sum=0;
    int temp1=n,temp2=1;
    if(a==0 || a==n)
    {
        return 1;
    }

    for(int i=1;i<a;i++)
    {
        temp1=temp1*(--n);
    }
    for(int i=1;i<=a;i++)
    {
        temp2=temp2*i;
    }
    sum=temp1/temp2;
    return sum;
}

```

4. 效果图

图 13-3 是二次、三次和 N 次贝塞尔曲线的效果图展示，其中第一列是二次贝塞尔曲线，第二列是三次贝塞尔曲线，第三列是 N 次贝塞尔曲线。每一列的控制点数量相同，但是位置不同，从而得到不同的曲线。红色线段是控制点之间的直线连接，蓝色曲线是由控制点生成的贝塞尔曲线。

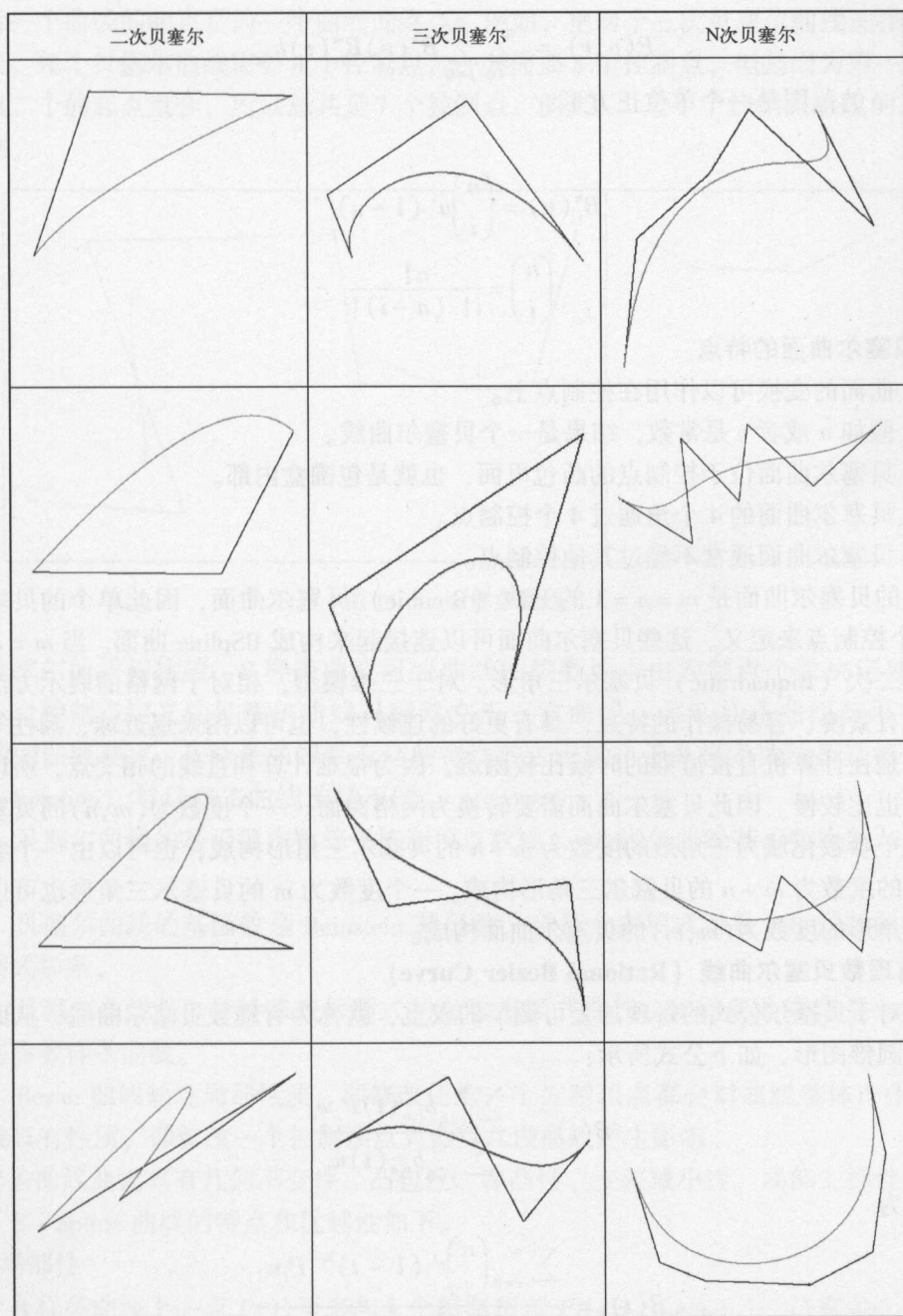


图 13-3 各种贝塞尔曲线

13.2.5 贝塞尔曲面

一个阶数为 (m, n) 的贝塞尔曲面可以由 $(n+1)(m+1)$ 个控制点构造。贝塞尔曲面把单位正方形映射到和控制点维数相同的曲面上。一个具有两个参数 u, v 的贝塞尔是参数化曲面，其中曲面上一个点 P 的位置为

$$P(u, v) = \sum_{i=0}^n \sum_{j=0}^m B_i^n(u) B_j^m(v) k_{i,j}$$

式中， u 、 v 的范围是一个单位正方形。

其中，

$$B_i^n(u) = \binom{n}{i} u^i (1-u)^{n-i}$$

$$\binom{n}{i} = \frac{n!}{i! (n-i)!}$$

1. 贝塞尔曲面的特点

- (1) 曲面的变换可以作用在控制点上。
- (2) 假如 u 或者 v 是常数，结果是一个贝塞尔曲线。
- (3) 贝塞尔曲面位于控制点的凸包里面，也就是包围盒内部。
- (4) 贝塞尔曲面的 4 个角通过 4 个控制点。
- (5) 贝塞尔曲面通常不经过其他控制点。

常用的贝塞尔曲面是 $m=n=3$ 的三次（Bicubic）贝塞尔曲面，因此单个的贝塞尔曲面需要 16 个控制点来定义。这些贝塞尔曲面可以连接起来构成 BSpline 曲面。当 $m=n=2$ 时，可以生成二次（Biquadratic）贝塞尔三角形。对于三维模型，相对于网格的表示方法，贝塞尔曲面具有紧凑、容易操作的特点，具有更好的连续性，也可以用来逼近球、圆柱等参数化曲面。但是在计算机直接渲染的时候比较困难，因为很难计算和直线的相交点，所以用光线追踪算法也比较慢。因此贝塞尔曲面需要转换为网格曲面。一个度数为 (m, n) 的贝塞尔曲面可以由两个参数化域为三角形的度数为 $m+n$ 的贝塞尔三角形构成，也可以由一个参数化域为正方形的度数为 $m+n$ 的贝塞尔三角形构成。一个度数为 m 的贝塞尔三角形也可以由参数化域为三角形的度数为 (m, m) 的贝塞尔曲面构成。

2. 有理数贝塞尔曲线（Rational Bezier Curve）

如果对于贝塞尔公式的每项加上可调节的权重，就称为有理数贝塞尔曲线，从而可以表示复杂的圆锥图形，如下公式所示：

$$B(t) = \frac{\sum_{i=0}^n b_{i,n}(t) P_i w_i}{\sum_{i=0}^n b_{i,n}(t) w_i}$$

或者展开为

$$B(t) = \frac{\sum_{i=0}^n \binom{n}{i} t^i (1-t)^{n-i} P_i w_i}{\sum_{i=0}^n \binom{n}{i} t^i (1-t)^{n-i} w_i}$$



13.3 B-Spline 曲线

13.3.1 B 样条曲线特点

B 样条（B-Spline 曲线）指的是把多个贝塞尔曲线连接起来得到的曲线。在 B 样条曲

线中, 后一个曲线的起点是前一个曲线的终点。例如, 把两个三次贝塞尔曲线连接为一个 B 样条曲线, 每个贝塞尔曲线需要 4 个控制点, 一共需要 8 个控制点, 但是因为第一个曲线的终点和第二个的起点重合, 所以总共是 7 个控制点。图 13-4 是 4 个控制点生成的 B 样条曲线的实例。

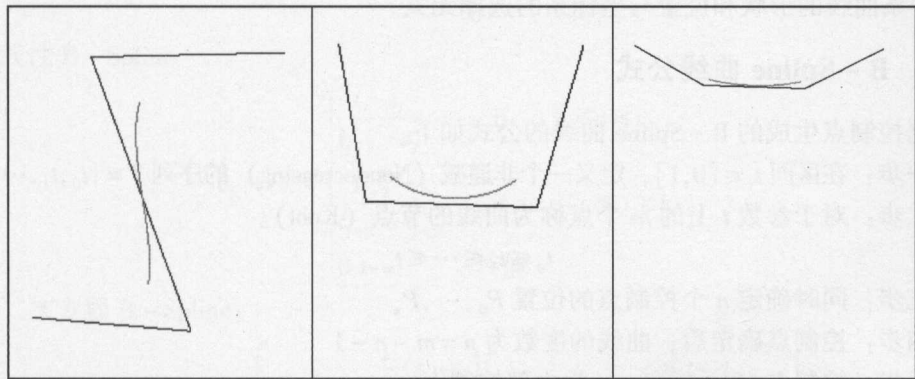


图 13-4 B 样条曲线

与贝塞尔曲线相比较, B 样条曲线可将曲线的阶数原本由控制点个数决定独立出来, 因此由 4 个控制点定义的贝塞尔曲线只能建立出三次曲线, 而 B 样条曲线却可有一次至三次不同的曲线建立。B 样条曲线具有这些特性的主因就在于 B 样条曲线所选择的基底函数 (Basis function) 与贝塞尔曲线选择不同。

(1) 贝塞尔曲线的基函数次数等于控制顶点数减 1, B 样条曲线基函数次数与控制顶点数无关。

(2) 贝塞尔曲线的基函数是 Beinstein 基函数, 它是个多项式函数, B 样条曲线的基函数是多项式样条。

(3) 贝塞尔曲线是一种特殊表示形式的参数多项式曲线, B 样条曲线则是一种特殊表示形式的参数样条曲线。

(4) Bezier 曲线缺乏局部性质, 即修改任意一个控制顶点都会对曲线整体产生影响。B 样条曲线具有性质, 即修改一个控制顶点只会对几段曲线产生影响。

B 样条曲线曲面具有几何不变性、凸包性、保凸性、变差减小性、局部支撑性等许多优良性质。B-Spline 曲线的特点和优越性如下。

1) 局部性

k 阶 B 样条曲线上一点 $P(t)$ 至多与 k 个控制顶点 $P_j (j = i - k + 1, \dots, i)$ 有关, 与其他控制顶点无关; 移动该曲线的第 i 个控制顶点 P_i 至多影响定义在区间 $(t_i, t_i + k)$ 上那部分曲线的形状, 对曲线的其余部分不发生影响。

2) 凸包性

k 阶 $P(t)$ 在区间 (t_i, t_{i+1}) 上的部分位于 k 个点 P_{i-k+1}, \dots, P_i 的凸包内, 整条曲线则位于各凸包 C_i 的并集之内。

3) 分段参数多项式

分段参数多项式 $P(t)$ 在每一区间上都是次数不高于 $k-1$ 的参数 t 的多项式。

4) 变差缩减性

设平面内 $n+1$ 个控制顶点构成 B 样条曲线 $P(t)$ 的特征多边形。在该平面内的任意一条直线与 $P(t)$ 的交点个数不多于该直线和特征多边形的交点个数。

5) 几何不变性

B 样条曲线的形状和位置与坐标系的选择无关。

13.3.2 B-Spline 曲线公式

根据控制点生成的 B-Spline 曲线的公式如下。

第一步：在区间 $t_i \in [0, 1]$ ，定义一个非递减 (Nondecreasing) 的序列 $T = \{t_0, t_1, \dots, t_m\}$ 。

第二步：对于参数 t 上的 m 个点称为曲线的节点 (Knot)。

$$t_0 \leq t_1 \leq \dots \leq t_{m-1}$$

第三步：同时确定 n 个控制点的位置 P_0, \dots, P_n

第四步：控制点确定后，曲线的度数为 $p = m - n - 1$

第五步：控制点 $t_{p+1}, \dots, t_{m-p-1}$ 为内部控制点。

第六步：度数为 n 的 B-Spline 曲线把辉哥参数区间映射到一个实数，表示为 $S: [t_n, t_{m-n-1}] \rightarrow R$

第七步：基本函数递归定义如下。

$$b_{j,0}(t) = \begin{cases} 1 & \text{if } t_j \leq t < t_{j+1}, \quad j=0, \dots, m-2 \\ 0 & \text{其他} \end{cases}$$

$$b_{j,n}(t) = \frac{t - t_j}{t_{j+n} - t_j} b_{j,n-1}(t) + \frac{t_{j+n+1} - t}{t_{j+n+1} - t_{j+1}} b_{j+1,n-1}(t), \quad j=0, \dots, m-n-2$$

第八步：由基本 (Basis Function) 构成的 B-Spline 曲线公式为

$$S(t) = \sum_{i=0}^{m-n-2} P_i b_{i,n}(t), \quad t \in [t_n, t_{m-n-1}]$$

第九步：假如节点是等距的，那么构成的 B-Spline 是一致 (Uniform) B-Spline，否则称为非一致 (Non-Uniform) B-Spline。

第十步：在基本函数中，满足下面的公式。

$$b_{j,n}(t) = b_n(t - t_j), \quad j=0, \dots, m-n-2$$

第十一步：其中，

$$b_n(t) = \frac{n+1}{n} \sum_{i=0}^{n+1} \omega_{i,n} (t - t_i)^n$$

$$\omega_{i,n} = \prod_{j=0, j \neq i}^{n+1} \frac{1}{t_j - t_i}$$

$$(t - t_i)_+^n = \begin{cases} (t - t_i)^n & \text{if } t \leq t_i \\ 0 & \text{if } t < t_i \end{cases}$$

第十二步：如果控制点比度数多 1，那么 B-Spline 就是一个贝塞尔曲线。每个基本 B-Spline 只在一个曲线内为非零值。

$$b_{i,n}(t) = \begin{cases} >0 & \text{if } t \leq t_{i+n+1} \\ 0 & \text{其他} \end{cases}$$

上面的公式是通用的 B 样条曲线计算公式, 针对控制点数量的不同, B 样条曲线单独的公式展开如下所示。

1) 常数 B - Spline

$$b_{j,0}(t) = 1_{[t_j, t_{j+1}]} = \begin{cases} 1 & \text{if } t_j \leq t < t_{j+1} \\ 0 & \text{其他} \end{cases}$$

2) 线性 B - Spline

$$b_{j,1}(t) = \begin{cases} \frac{t - t_j}{t_{j+1} - t_j} & \text{if } t_j \leq t < t_{j+1} \\ \frac{t_{j+2} - t}{t_{j+2} - t_{j+1}} & \text{if } t_{j+1} \leq t < t_{j+2} \\ 0 & \text{其他} \end{cases}$$

3) 二次方程 B - Spline:

$$b_{j,2}(t) = \begin{cases} \frac{1}{2}(t - t_j)^2 & t_j \leq t \leq t_{j+1} \\ -(t - t_{j+1})^2 + (t - t_{j+1}) + \frac{1}{2} & t_{j+1} \leq t \leq t_{j+2} \\ \frac{1}{2}(1 - (t - t_{j+2}))^2 & t_{j+2} \leq t \leq t_{j+3} \\ 0 & \text{其他} \end{cases}$$

变换为矩阵形式为

$$S_i(t) = [t^2 \quad t \quad 1] \frac{1}{2} \begin{bmatrix} 1 & 2 & 1 \\ -2 & 2 & 0 \\ 1 & 1 & 0 \end{bmatrix} \begin{bmatrix} P_{i-1} \\ P_i \\ P_{i+1} \end{bmatrix}$$

4) 三次方 B - Spline 公式

$$S_i(t) = [t^3 \quad t^2 \quad t \quad 1] \frac{1}{6} \begin{bmatrix} -1 & 3 & -3 & 1 \\ 3 & -6 & 3 & 0 \\ -3 & 0 & 3 & 0 \\ 1 & 4 & 1 & 0 \end{bmatrix} \begin{bmatrix} P_{i-1} \\ P_i \\ P_{i+1} \\ P_{i+2} \end{bmatrix}$$

13.3.3 B 样条曲线代码

B 样条曲线的代码可以根据公式直接进行计算, 但是和贝塞尔曲线一样, 对不同数量的控制点需要用不同的函数。每种度数的 B 样条曲线也分为两步, 第一步生成控制点, 控制点位置是随机的, 可以手工调整, 第二步根据上述公式计算 B 样条曲线上点的位置。

1. 四点 B 样条曲线

1) 生成四点 B 样条曲线控制点

```
public TriMesh CreateFourPointBSpline()
{
    TriMesh BezierControlPoint = new TriMesh();
```



```
BezierControlPoint.Vertices.Add(new VertexTraits(0,0,0));
BezierControlPoint.Vertices.Add(new VertexTraits(0.2,0.2,0));
BezierControlPoint.Vertices.Add(new VertexTraits(0.4,0.5,0));
BezierControlPoint.Vertices.Add(new VertexTraits(0.6,0.4,0));
return BezierControlPoint;
}
```

2) 生成四点 B 样条曲线

```
public TriMesh CreateFourPointBSplineCurve(TriMesh mesh)
{
    TriMesh FourPointBSplineCurve = new TriMesh();
    double x=0;
    double y=0;
    double z=0;
    for(int t=0;t<=VerticesNum;t++)
    {
        double tt=(double)t/(double)VerticesNum;
        x=(double)1/6*((-Math.Pow(tt,3)+3*Math.Pow(tt,2)-3*tt+1))
            *mesh.Vertices[0].Traits.Position.x
            +(double)1/6*((3*Math.Pow(tt,3)-6*Math.Pow(tt,2)+4))
            *mesh.Vertices[1].Traits.Position.x
            +(double)1/6*((-3*Math.Pow(tt,3)+3*Math.Pow(tt,2)+3*tt+1))
            *mesh.Vertices[2].Traits.Position.x
            +(double)1/6*Math.Pow(tt,3)*mesh.Vertices[3].Traits.Position.x;
        y=(double)1/6*((-Math.Pow(tt,3)+3*Math.Pow(tt,2)-3*tt+1))
            *mesh.Vertices[0].Traits.Position.y
            +(double)1/6*((3*Math.Pow(tt,3)-6*Math.Pow(tt,2)+4))
            *mesh.Vertices[1].Traits.Position.y
            +(double)1/6*((-3*Math.Pow(tt,3)+3*Math.Pow(tt,2)+3*tt+1))
            *mesh.Vertices[2].Traits.Position.y
            +(double)1/6*Math.Pow(tt,3)*mesh.Vertices[3].Traits.Position.y;
        FourPointBSplineCurve.Vertices.Add(new VertexTraits(x,y,z));
    }
    return FourPointBSplineCurve;
}
```

2. N 点 B 样条曲线

1) 生成 N 点 B 样条曲线控制点

```
public TriMesh CreateNBSplineControlPoint()
{
    TriMesh cPoint = new TriMesh();
    Random r = new Random();
}
```

```

cPoint.Vertices.Add(new VertexTraits(0,0,0));
for(int t=0;t<Point;t++)
{
    int x=r.Next(1,10);
    cPoint.Vertices.Add(new VertexTraits(t*0.1+x*0.01,
                                          x*0.1,0));
}
return cPoint;
}

```

2) 生成 N 点 B 样条曲线

```

public TriMesh CreateNBsplineCurve(TriMesh mesh)
{
    TriMesh BezierCurve = new TriMesh();
    double x=0;
    double y=0;
    double z=0;

    for(int i=0;i<Point-2;i++)
    {
        for(int t=0;t<=VerticesNum;t++)
        {
            double tt=(double)t/(double)VerticesNum;
            x=(double)1/6*((-Math.Pow(tt,3)+3*Math.Pow(tt,2)-3*tt+1))
              *mesh.Vertices[i].Traits.Position.x
              +(double)1/6*((3*Math.Pow(tt,3)-6*Math.Pow(tt,2)+4))
              *mesh.Vertices[i+1].Traits.Position.x
              +(double)1/6*((-3*Math.Pow(tt,3)+3*Math.Pow(tt,2)+3*tt+1))
              *mesh.Vertices[i+2].Traits.Position.x
              +(double)1/6*Math.Pow(tt,3)*mesh.Vertices[i+3].Traits.Position.x;
            y=(double)1/6*((-Math.Pow(tt,3)+3*Math.Pow(tt,2)-3*tt+1))
              *mesh.Vertices[i].Traits.Position.y
              +(double)1/6*((3*Math.Pow(tt,3)-6*Math.Pow(tt,2)+4))
              *mesh.Vertices[i+1].Traits.Position.y
              +(double)1/6*((-3*Math.Pow(tt,3)+3*Math.Pow(tt,2)+3*tt+1))
              *mesh.Vertices[i+2].Traits.Position.y
              +(double)1/6*Math.Pow(tt,3)*mesh.Vertices[i+3].Traits.Position.y;
            BezierCurve.Vertices.Add(new VertexTraits(x,y,z));
            x=0;
            y=0;
        }
    }
}

```

```
return BezierCurve;
```

3. 效果图

图 13-5 是 10 个控制点的 B 样条曲线和 6 个控制点的 B 样条曲线的效果图，每列的控制点数量一样，但是位置不一样，从而生成不同的曲线，其中蓝色的曲线是 B 样条曲线，红色的线段是连接控制点的直线。

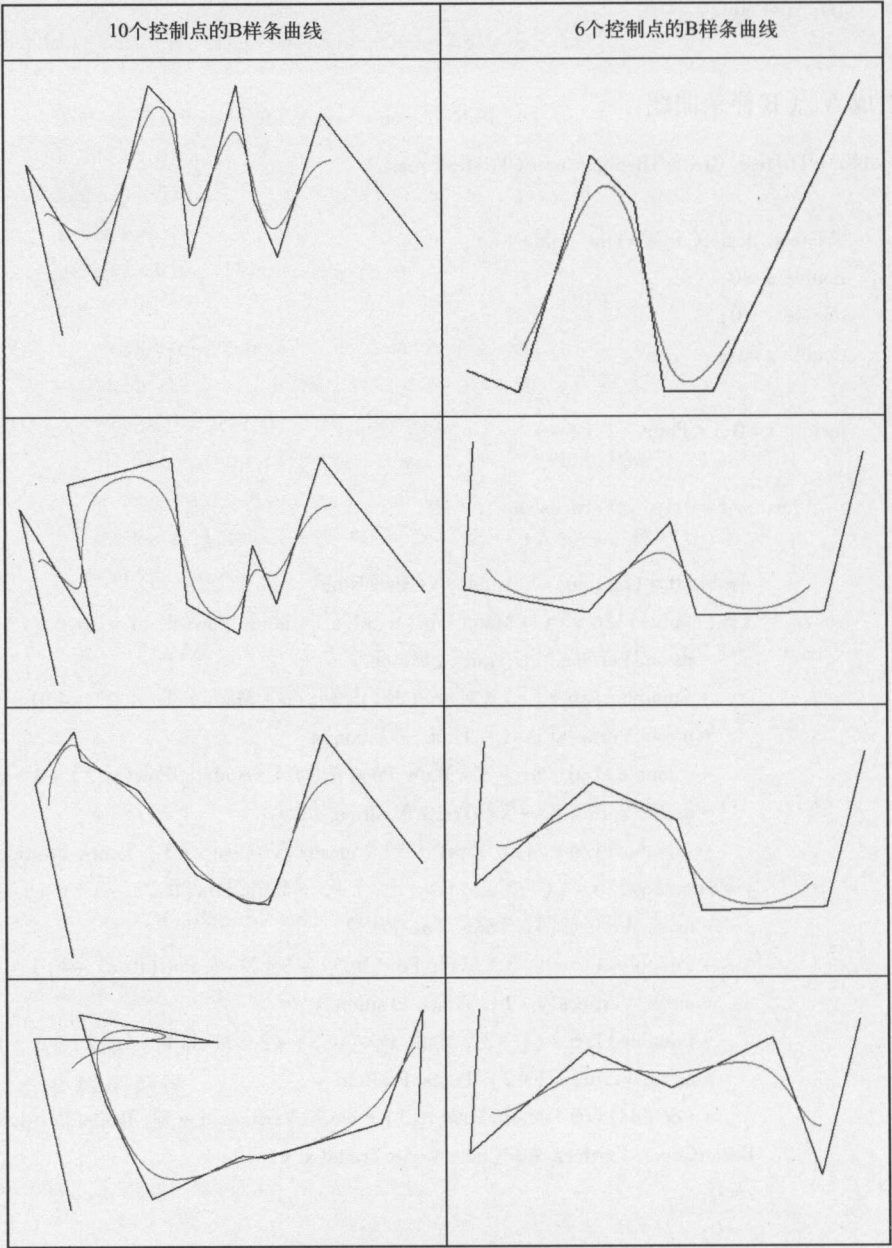


图 13-5 10 个控制点的 B 样条曲线和 6 个控制点 B 样条曲线



13.4 NURBS 曲线

13.4.1 定义和属性

NURBS (Non - uniform rational B - spline) 曲线可以用来生成三维曲线和曲面。NURBS 是非均匀有理 B 样条曲线 (Non - Uniform Rational B - Splines) 的缩写。贝塞尔曲线、有理贝塞尔曲线、均匀 B 样条和非均匀 B 样条都被统一到 NURBS 曲线中。NURBS 曲线通过权重或者控制节点提供高级的局部曲线控制, NURBS 曲线允许样条曲线的一部分被修改时不会影响另外一部分。权重与每一个控制点相结合, 它们指定控制点与曲线定点之间的距离。默认设置下, 样条上所有控制点具有相同的权重因子, 但是也可以修改曲线的权重。处理 NURBS 曲线的权重可以改善一条曲线的细微形状, 但通常也减慢了最后模型的渲染速度。

1. NURBS 曲线属性

- (1) 在仿射变换 (Affine Transformation) 下保持不变、旋转、位移等操作可以直接作用在控制点上。
- (2) 对于分析 (Analytic) 形状和自由 (Free - Form) 形状的曲面数学公式都一致。
- (3) 可以灵活地设计各种类型的形状。
- (4) 相对其他的曲面表示方法, 减少了内存需求。
- (5) 具有稳定和精确的数值计算方法。

NURBS 曲线实例如图 13-6 所示, 其中蓝色的曲线是 NURBS 曲线, 红色线段是控制点之间的连接线。

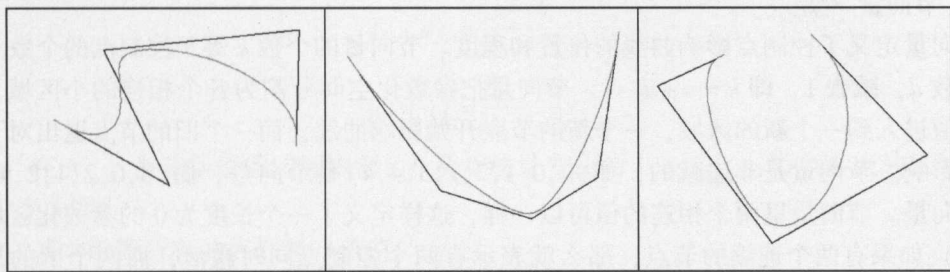


图 13-6 NURBS 曲线实例

NURBS 曲线和曲面起源于船体、飞行器、汽车等的外形设计的精确数学表示。NURBS 曲线是一个参数的函数, 而 NURBS 曲面是两个参数的函数。曲面的形状由控制点决定, NURBS 曲面能够表示简单的形状, 更复杂的曲面需要细分曲面来表示。这两种曲面需要的控制点都比较少。NURBS 曲面的控制点比较直观, 生成的效果是可预期的。一个复杂的三维模型可以由几个 NURBS 曲面构成。这些曲面在相交处的几何连续性 (Geometry Continuity) 有如下几种。

- (1) 位置连续 G0 (Positional Continuity): 指的是两个曲面在边界相连, 但是相连处的角度可能改变很大。
- (2) 切线连续 G1 (Tangential Continuity): 指的是两个曲面在边界不仅仅几何位置一

致，而且相应的切向量也一致。

(3) 曲率连续 G_2 (Curvature Continuity): 指的是在切线连续的基础上，连接处的向量不仅仅一致，而且向量的变化也要一致，也就是曲率要一致。这样的连续性使两个曲面看起来和一个曲面一样。这种连续性在视觉上是光滑的。

NURBS 曲面是函数，因此可以定义此曲面相对于参数的导数 (C_n)。曲线的参数化连续性 C_1 和 G_1 、 C_2 和 G_2 是一致的，但是 C_3 比 G_3 约束要小，也就是满足 C_3 不一定满足 G_3 。一个 n 阶连续的曲线，需要能够在节点处 n 阶可求导。

2. NURBS 曲线的结构

一个 NURBS 曲线由阶数 (Order)、有权重的控制点、结向量 (knot vector) 来定义。NURBS 曲面和曲面是 B-Spline 和 Bezier 曲面与曲面的扩展。最要的区别是控制点的权重，从而使 NURBS 曲线变为有理数的 (Rational)，而 B-Spline 是一种特殊的 NURBS。通过计算一个和两个参数上 NURBS 公式的值，可以得到三维空间的贝塞尔曲线和曲面。

1) 权重

控制点决定了曲线的形状，曲线上每个点的位置由相对应的一些控制点的有权重的和来计算。每个控制点的权重随着参数进行变化。对于一个度数为 d 的曲线，每个控制点的权重只在由节点向量表示的 $d+1$ 个参数区域内为非零值，其余区域都是零值。这个非零值的权重取决于一个度数为 d 的多项式函数，称为基本函数 (Basis Function)。在区域的边界，权重光滑变化为零。例如，对于度数为 1 的线性基本函数是一个三角函数。此函数从 0 线性变为 1，又从 1 线性变为零。一个控制点的权重只在参数化区域的某个间隔为非零值，所以只影响曲线上相对应的某一段曲线，也就是局部曲线，这样移动这个控制点，只改变曲线的局部形状，而不改变其他部分的形状。

2) 节向量

节向量定义了控制点影响曲线的位置和程度。节向量的个数 k 等于控制点的个数 n 加曲线的度数 d ，减去 1，即 $k = n + d - 1$ 。节向量把参数化空间分割为各个相连的小区域。每次参数化值进入到一个新的区域，一个新的节点开始影响曲线，而一个旧的节点退出对下一段曲线的影响。节向量是非递减的，如 $(0, 0, 1, 2, 2, 3, 4, 4)$ 是节向量，而 $(0, 0, 2, 1, 2, 4, 4, 3)$ 不是节向量。节向量里两个相连的值可以一样，这样定义了一个长度为 0 的参数化区域。

(1) 如果有两个连续的节点，那么就表示有两个控制点同时激活，而两个旧的控制点就退出。

(2) 这样会对曲线的连续性和高阶连续性造成影响，并且可以在曲线上生成一个角 (Corner)。

(3) 相同的节点数量称为节点的乘数 (Multiplicity)。

(4) 节点的乘数受到曲线的度数限制。

(5) 一个更高乘数的节点，会把此曲线分为两个不相连的部分。

(6) 对于度数为 1 的曲线，每个节点对应一个控制点。

(7) 节点向量开始的节点乘数通常和曲线的阶数一致，如阶数为 3 的曲线，开始的 3 个节点值一样。

(8) 节点向量结尾的节点乘数通常和曲线的阶数一致，这样曲线的开始和终止于两个控制点。

(9) 节点的具体值不重要, 关键是值的比例, 如(0,0,1,2,3,3)和(0,0,3,6,9,9)两个节向量构成的曲线是一样的。

(10) 节点的位置影响参数化区域到曲线的映射。

(11) 节点向量通常用于内部计算, 而不用用于外部交互设计时显示。

3) 阶数 (Order)

阶数定义了曲线上任意一个点受到影响的控制点数量。阶数值为曲线的度数加1。例如, 二阶的曲线也就是度数为1的线性曲线, 三阶的曲线也就是度数为2的二次曲线。控制点的数量必须大于等于阶数。

13.4.2 NURBS 曲线公式

NURBS 曲线计算公式也是由基本函数通过权重构成的。在定义了基本函数之后, 根据权重把这些基本函数混合到一起就得到了 NURBS 曲线的计算公式。

1. 基本函数 (Basis Function)

(1) 曲线的基本函数用 $N_{i,n}(u)$ 来表示。其中, i 表示第 i 个控制点, n 表示基本函数的度数。

(2) $N_{i,0}$ 是一个分段常数函数, 即在一个参数化区域为1, 其余为0。

(3) $N_{i,n}$ 是由两个低次的函数 $N_{i,n-1}$ 和 $N_{i+1,n-1}$ 线性插值构成的。

(4) $N_{i,n-1}$ 和 $N_{i+1,n-1}$ 在 n 个参数化区域内为非零值, 并且在 $n-1$ 个参数化区域内重叠。

(5) $N_{i,n}$ 计算公式如下。

$$N_{i,n} = f_{i,n} N_{i,n-1} + g_{i+1,n} N_{i+1,n-1}$$

其中

$$f_{i,n}(u) = \frac{u - k_i}{k_{i+n} - k_i}$$

$$g_{i,n}(u) = \frac{k_{i+n} - u}{k_{i+n} - k_i}$$

(6) 在 $N_{i,n-1}$ 为非零值的区域内部, f_i 从0线性变化为1, $N_{i+1,n-1}$ 为非零值的区域内部, g_{i+1} 从1变为零。

(7) 基本函数所有的值都是非负值。

(8) 对于任意一个参数, 基本函数的和为1。

(9) 如果一个节点区域的长度和其他相比非常小, 那么此处的基本函数为行程一个尖峰。那么生成的曲线就逼近这个控制点, 如果两个节点相同, 那么曲线在此处形成一个角。

2. NURBS 曲线的公式

$$C(u) = \frac{\sum_{i=1}^k \frac{N_{i,n} w_i}{\sum_{j=1}^k N_{j,n} w_j} P_i}{\sum_{i=1}^k \frac{N_{i,n} w_i}{\sum_{j=1}^k N_{j,n} w_j}} = \frac{\sum_{i=1}^k N_{i,n} w_i P_i}{\sum_{i=1}^k N_{i,n} w_i}$$

K 是控制点的个数, w_i 是权重, 分母是一个正则化的量, 加入所有权重为1, 那么分母为1。如果加上权重, 那么下面公式表示有理基本函数 (Rational Basis Function):

$$R_{i,n}(u) = \frac{N_{i,n}(u) w_i}{\sum_{j=1}^k N_{j,n}(u) w_j}$$

每个控制点除了混合函数还有权重（Weight），权重表示此控制点吸引曲线的程度，权重的绝对值不重要，重要的是各个控制点权重的相对大小。把所有权重设置为 1 和把所有权重设置为 100 的曲线形状是一样的。贝塞尔曲线可以看作是没有权重的特殊的 NURBS 曲线。

3. NURBS 曲线的特点

- (1) N 个控制点的曲线可以变为 $N+1$ 个控制点且相同形状的曲线。
- (2) 需要从中生成新的 $N+1$ 个控制点和插入一个新的节点。
- (3) 通常用于交互式设计，在插入一个新控制点后，曲线和原来的形状一样，这样可以用于之后的曲线设计。
- (4) 节点插入操作把一个新的节点插入到节点向量里面，可以插入多次，直到满足节点的乘数。
- (5) 节点的删除操作把一个节点删除，但是这个操作可能造成曲线的形状改变。
- (6) 度数升级操作，一个底度数的曲线可以用高度数的曲线表示，这种操作用于把两个不同度数的曲线用高度数的曲线连接起来。
- (7) 曲线的曲率公式为

$$\kappa = \frac{|r'(t) \times r''(t)|}{|r'(t)|^3}$$

13.4.3 NURBS 曲线代码

NURBS 曲线生成函数可以根据公式直接计算得到，输入的是控制点的位置，输出的是 NURBS 曲线上的采样点。把这些采样点连接起来就得到完整的 NURBS 曲线。

- (1) 生成 NURBS 曲线。

```
public TriMesh CreateNURBS(TriMesh mesh)
{
    TriMesh nurbs = new TriMesh();
    double x = 0;
    double y = 0;
    double z = 0;
    double m1 = 0, m2 = 0;
    double h = 0;
    double[] w = {0.9, 0.3}; // 权值
    double[] ut = new double[4000];
    for(int i = 0; i < VerticesNums; i++)
    {
        if(i >= 3999)
            break;
        ut[i] = (double)((double)i / (double)2000) * 7;
    }
    for(int t = 0; t < VerticesNums; t++)
    {
        double u = (double)t / (double)2000;
```

```
for(int i=0;i < all + 1;i++)
{
    m1 = m1 + mesh. Vertices[i]. Traits. Position. x * CN(i,3,u,ut) * w[i%2];
    m2 = m2 + mesh. Vertices[i]. Traits. Position. y * CN(i,3,u,ut) * w[i%2];
}
for(int i=0;i < all + 1;i++)
{
    h = h + CN(i,3,u,ut) * w[i%2];
}
x = m1/h;
y = m2/h;
nurbs. Vertices. Add(new VertexTraits(x,y,z));
m1 =0;m2 =0;h =0;
}
return nurbs;
}
```

(2) NURBS 曲线可以表示很多特殊的曲线，如可以表示一个圆，表示方法不唯一，其中一种方法需要设置的控制点位置和权重参数是：

x	y	z	Weight
1	0	0	1
1	1	0	$\sqrt{2}/2$
0	1	0	1
-1	1	0	$\sqrt{2}/2$
-1	0	0	1
-1	-1	0	$\sqrt{2}/2$
0	-1	0	1
1	-1	0	$\sqrt{2}/2$
1	0	0	1

节点向量为：
 $\{0,0,0,\pi/2,\pi/2,\pi,\pi,3\pi/2,3\pi/2,2\pi,2\pi,2\pi\}$
这是一个阶数为 3、度数为 2 的二次曲线。圆由 4 个四分圆组成，两个四分圆由重复两次的节点连接到一起。通常两次重复节点对于度数为 2 的曲线来说会造成连续性丢失，但是由于控制点的特定位置，所以圆正好还可以连续。对于圆来说，每个点上都是无限可求导的。这个圆不是以弧长作为参数的，如除了开始端点、结束端点、中点、四分圆的端点以外，其他参数 t 生成的曲线点都不位于 $(\sin(t),\cos(t))$ 。如果参数 t 从 0 到 2π ，那么 NURBS 圆不会是一个整圆。

生成圆形曲线的代码如下。
① 生成控制点。

```
controlPoint.Vertices.Add(new VertexTraits(0.1,0,0));
controlPoint.Vertices.Add(new VertexTraits(0.1,0.1,0));
controlPoint.Vertices.Add(new VertexTraits(0,0.1,0));
controlPoint.Vertices.Add(new VertexTraits(-0.1,0.1,0));
controlPoint.Vertices.Add(new VertexTraits(-0.1,0,0));
controlPoint.Vertices.Add(new VertexTraits(-0.1,-0.1,0));
controlPoint.Vertices.Add(new VertexTraits(0,-0.1,0));
controlPoint.Vertices.Add(new VertexTraits(0.1,-0.1,0));
controlPoint.Vertices.Add(new VertexTraits(0.1,0,0));
```

② 生成 NURBS 圆形曲线。

```
public TriMesh CreateNURBSCircle(TriMesh mesh)
{
    TriMesh nurbs = new TriMesh();
    double x = 0;
    double y = 0;
    double z = 0;
    double m1 = 0, m2 = 0;
    double h = 0;
    double[] w = {0.9, 0.3}; // 权值
    double[] ut = new double[4000];
    for(int i = 0; i < VerticesNums; i++)
    {
        if(i >= 3999)
            break;
        ut[i] = (double)((double)i / (double)2000) * 7;
    }
    for(int t = 0; t < VerticesNums; t++)
    {
        double u = (double)t / (double)2000;
        for(int i = 0; i < circleAll + 1; i++)
        {
            m1 = m1 + mesh.Vertices[i].Traits.Position.x * CN(i, 3, u, ut) * w[i % 2];
            m2 = m2 + mesh.Vertices[i].Traits.Position.y * CN(i, 3, u, ut) * w[i % 2];
        }
        for(int i = 0; i < circleAll + 1; i++)
        {
            h = h + CN(i, 3, u, ut) * w[i % 2];
        }
        x = m1 / h;
        y = m2 / h;
        nurbs.Vertices.Add(new VertexTraits(x, y, z));
    }
}
```



```

        m1 = 0; m2 = 0; h = 0;
    }
    return nurbs;
}

```

(3) 椭圆 NURBS 曲线。

除了圆形曲线，NURBS 曲线还可以生成椭圆，生成椭圆的代码也分为设置控制点和计算曲线采样点两部分。

① 生成控制点。

```

controlPoint.Vertices.Add(new VertexTraits(0.1,0,0));
controlPoint.Vertices.Add(new VertexTraits(0.1,0.1,0));
controlPoint.Vertices.Add(new VertexTraits(-0.05,0.1,0));
controlPoint.Vertices.Add(new VertexTraits(-0.2,0.1,0));
controlPoint.Vertices.Add(new VertexTraits(-0.2,0,0));
controlPoint.Vertices.Add(new VertexTraits(-0.2,-0.1,0));
controlPoint.Vertices.Add(new VertexTraits(-0.05,-0.1,0));
controlPoint.Vertices.Add(new VertexTraits(0.1,-0.1,0));
controlPoint.Vertices.Add(new VertexTraits(0.1,0,0));

```

② 生成椭圆。

```

public TriMesh CreateNURBSEllipse( TriMesh mesh)
{
    TriMesh nurbs = new TriMesh();
    double x = 0;
    double y = 0;
    double z = 0;
    double m1 = 0, m2 = 0;
    double h = 0;
    double[] w = {0.9,0.5}; //权值
    double[] ut = new double[4000];
    for( int i = 0; i < VerticesNums; i++)
    {
        if(i >= 3999)
            break;
        ut[i] = (double)((double)i / (double)2000) * 7;
    }
    for( int t = 0; t < VerticesNums; t++)
    {
        double u = (double)t / (double)2000;
        for( int i = 0; i < ellipseAll + 1; i++)
        {
            m1 = m1 + mesh.Vertices[i].Traits.Position.x * CN(i,3,u,ut) * w[i%2];

```

```
        m2 = m2 + mesh. Vertices[i]. Traits. Position. y * CN(i,3,u,ut) * w[i%2];
    }
    for(int i=0;i < ellipseAll + 1;i++)
    {
        h = h + CN(i,3,u,ut) * w[i%2];
    }
    x = m1/h;
    y = m2/h;
    nurbs. Vertices. Add(new VertexTraits(x,y,z));
    m1 = 0;m2 = 0;h = 0;
}
return nurbs;
}
```

(4) 效果图

图 13-7 表示了圆形、椭圆和其他形状的 NURBS 曲线生成的图形。图中红色线段是连接控制点的直线，蓝色曲线是 NURBS 曲线。其中第一列是圆形，第二列是椭圆，第三列是其他形状曲线。调整控制点的位置，NURBS 曲线会发生改变，每一列的 NURBS 曲线控制点数量相同，但是每一行的控制点位置不同。

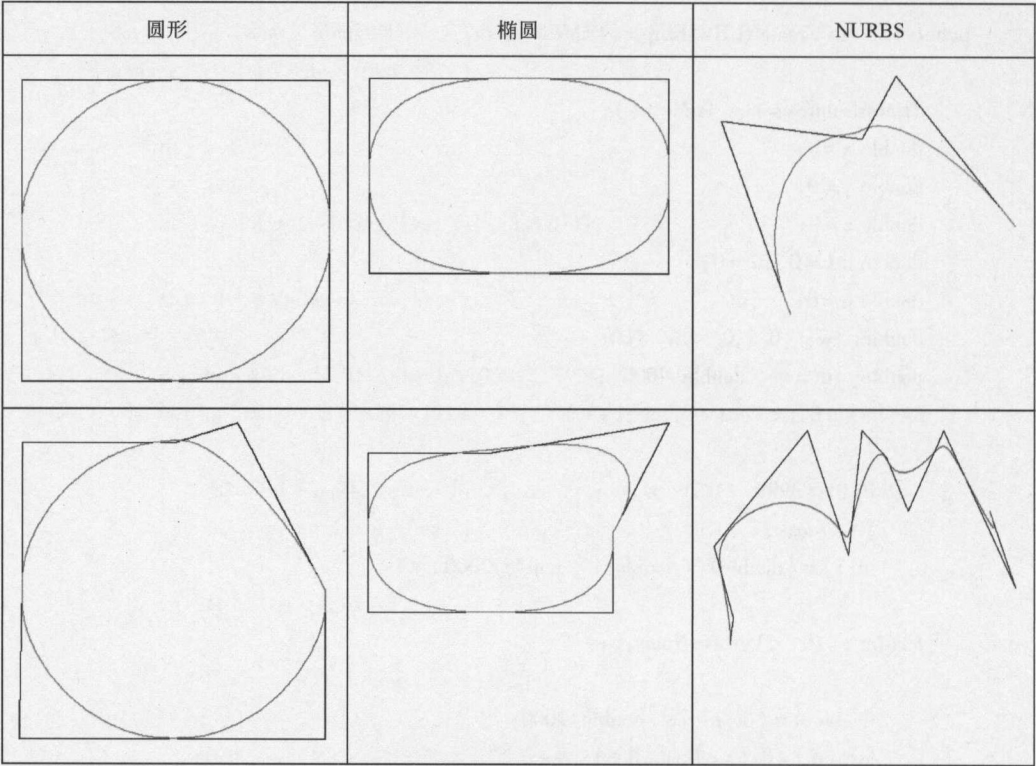


图 13-7 圆形、椭圆等 NURBS 曲线

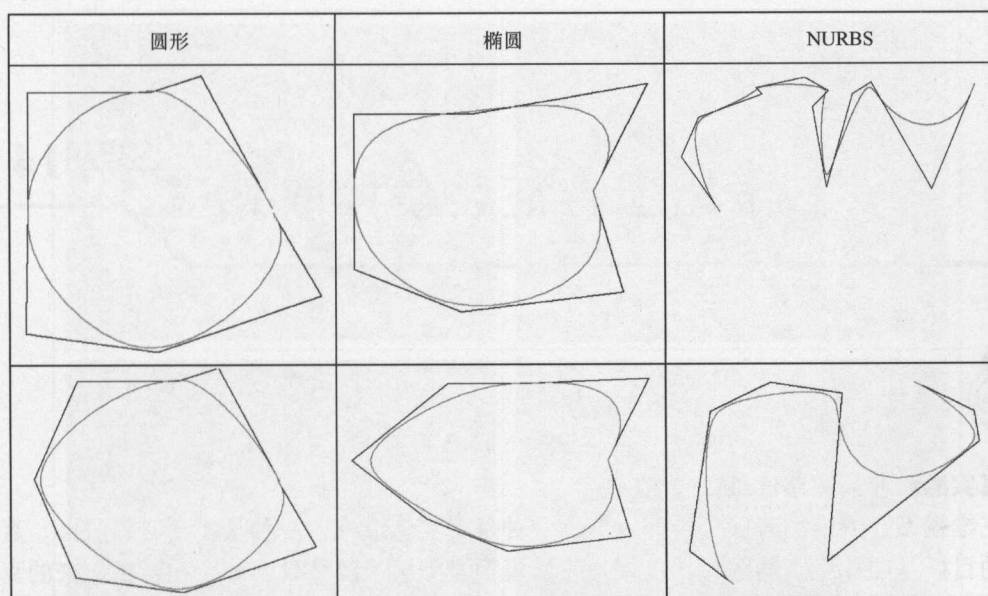


图 13-7 圆形、椭圆等 NURBS 曲线 (续)

第 14 章

三维模型特征线条抽取算法



14.1 Blender FreeStyle

真实感和非真实感渲染

三维模型在渲染的时候分为两大类，一种是真实感渲染，一种是非真实感渲染。真实感渲染的目标是照片级别的图片，也就是三维模型渲染出来的结果要和照相机拍出来的真实场景一样。但是在其他的许多应用中，追求的目标不一定是照片级别的图片。很多时候需要的是一张清晰简洁的示意图，或者是一幅融具有感情色彩的艺术画，还可能是一帧帧能够被快速绘制的动画，这些都称为非真实感渲染。非真实感绘制在可视化领域也非常重要，可以帮助更好地观察三维模型的相关特性，或者只通过绘制一些特殊线条达到加速交互可视化的目的。非真实感渲染追求的是有艺术创作效果，如水墨、铅笔画、卡通画、水彩画、油画等效果。通过特征线条描述物体是艺术创作的一个核心。真实感绘制技术在艺术上有一定的缺陷，从艺术家的角度看，一副图像除了对自然场景的模仿之外还应具有很强的表现力，而真实感绘制则难以做到。艺术家可以通过光照将图像中一些细节进行夸张或简化，从而引导观察者的视线到他感兴趣的区域，丰富图画的戏剧性效果。还可以使用不同画笔笔画表示物体表面的特征及光线在表面的反射，突出或忽略一些边界信息。非真实感绘制指的是利用计算机生成不具有照片般真实感，而具有手绘风格的图形的技术。其目标不在于图形的真实性，而主要在于表现图形的艺术特质、模拟艺术作品，或作为真实感图形的有效补充。非真实感可以认为是计算机辅助人们表达场景的一种方式，每一种特定的非真实感风格都能够满足人某些方面的需求。可以认为真实感只是非真实感的一种特殊形式。

图 14-1 是三维模型的真实感渲染效果。

图 14-2 是三维模型的非真实感渲染效果。

三维模型在显示的时候，很多时候需要显示三维模型的各种线条，如剪影、轮廓等。根据这些线条把三维模型显示出各种非真实感的效果。在非真实感的渲染效果中，大量使用三维模型上的这些特征线条。如何从三维模型中提取特征线条是三维模型处理的一个重要的操作。因此非真实感渲染分为两步，第一步是从三维模型上提取特征线条，第二步是设置参数，如颜色、粗细、风格等渲染这些线条。如图 14-3 所示，各种线条在抽取出来之后，可以用不同的颜色、粗细进行显示，从而得到不同的效果。

Blender 软件的 freestyle 模块支持非真实感的渲染，通过 freestyle，可以提取三维模型的各种特征线条，并且 freestyle 还提供界面来设置这些特征线条的显示参数。从而能够得到各种各样的非真实感的渲染效果。本章首先通过介绍 Blender 软件提取特征线条和渲染特征线

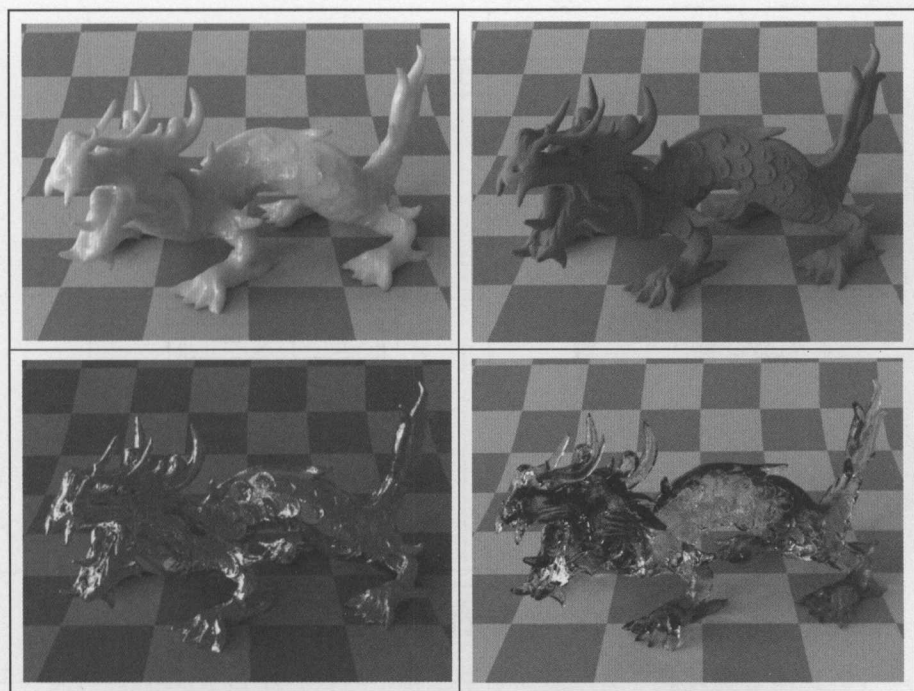


图 14-1 真实感渲染

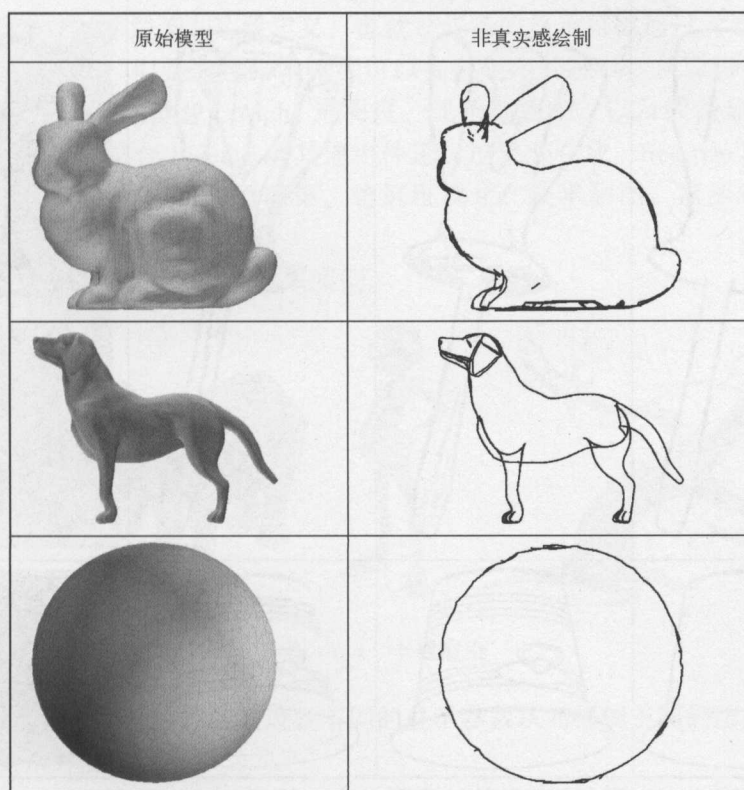


图 14-2 非真实感渲染

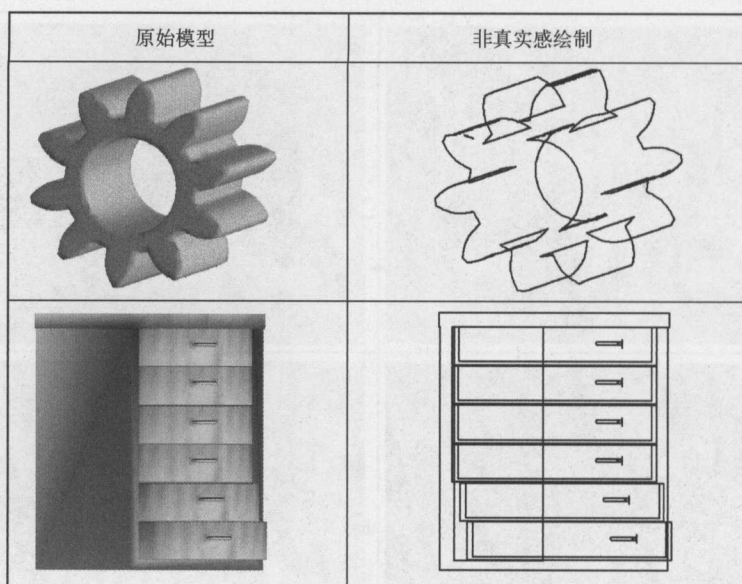


图 14-2 非真实感渲染（续）

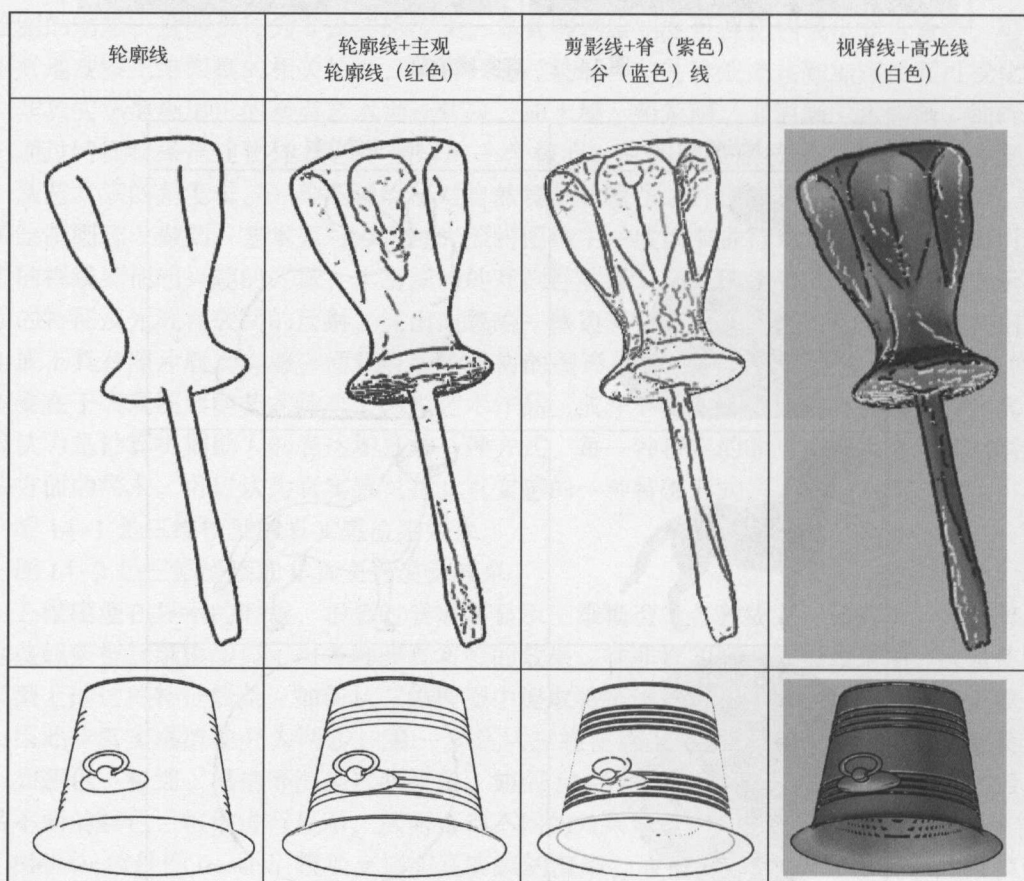


图 14-3 特征线条抽取和显示

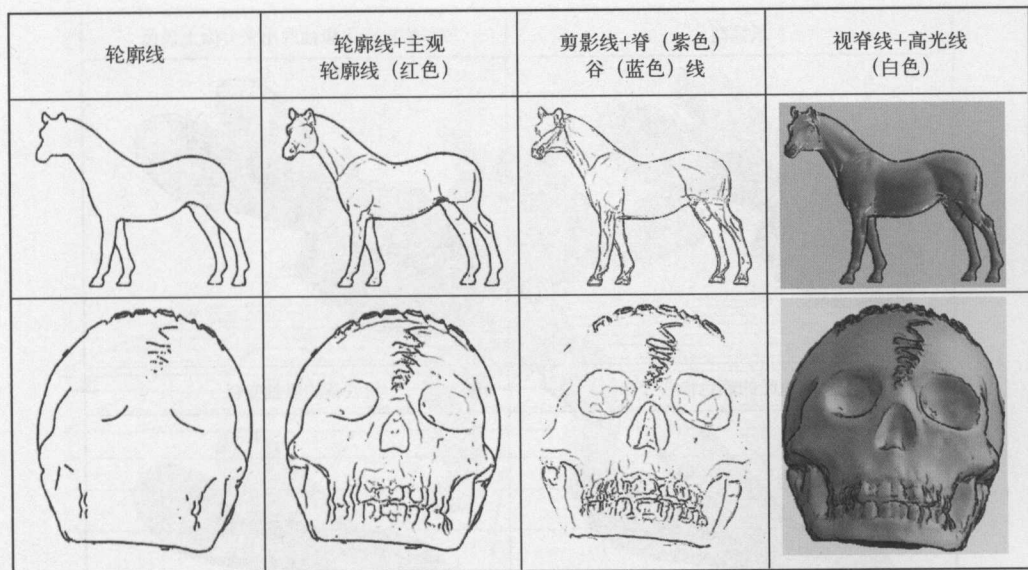


图 14-3 特征线条抽取和显示 (续)

条的应用界面, 通过 Blender 软件可以了解这些特征线条在非真实感中的应用, 从而对这些三维模型的特征线条有感性的认识, 然后再详细介绍各种线条的提取算法。freestyle 模块有两种工作模式: Python 脚本模式和参数编辑器模式。参数编辑模式拥有直观的编辑功能, 如虚线设置, 以及方便的多线型和边缘定义, 也就是线条的显示可以通过设置参数直接得到。脚本模式需要进行编程, 但是通过脚本编程可以提供更多的控制, 从而得到更强的渲染效果。freestyle 能调整线条的颜色、Alpha 透明度、线条的粗细, 以及线条显示的几何形状。最终产生的线条还可以结合 Blender 的其他组件进行渲染和合成。freestyle 的非真实感渲染功能可以用在各种应用中, 如卡通渲染、建筑可视化、技术制图、蓝图和计算机生成的草图。

图 14-4 是 freestyle 的卡通渲染效果实例。

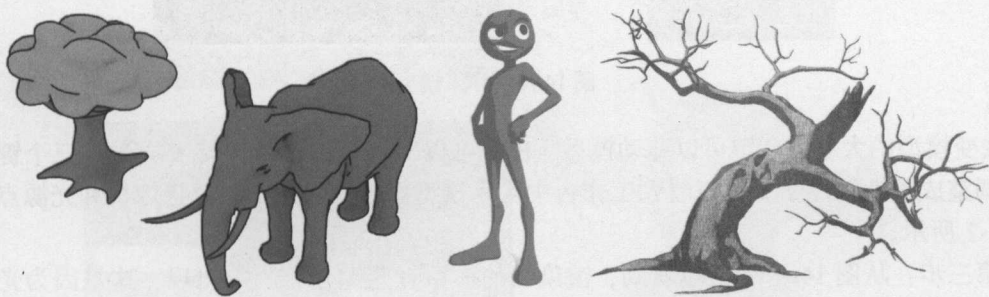


图 14-4 卡通渲染

图 14-5 是对于同样的特征线条设置不同的显示参数从而得到不同的渲染效果。

freeStyle 使用方法如下所示。

第一步: 通过菜单栏的文件选项导入事先准备好的模型, 如图 14-6 所示。

第二步: 将模型导入后, 其位置、大小可能与自己预期的不一样, 需要自己调整一下。

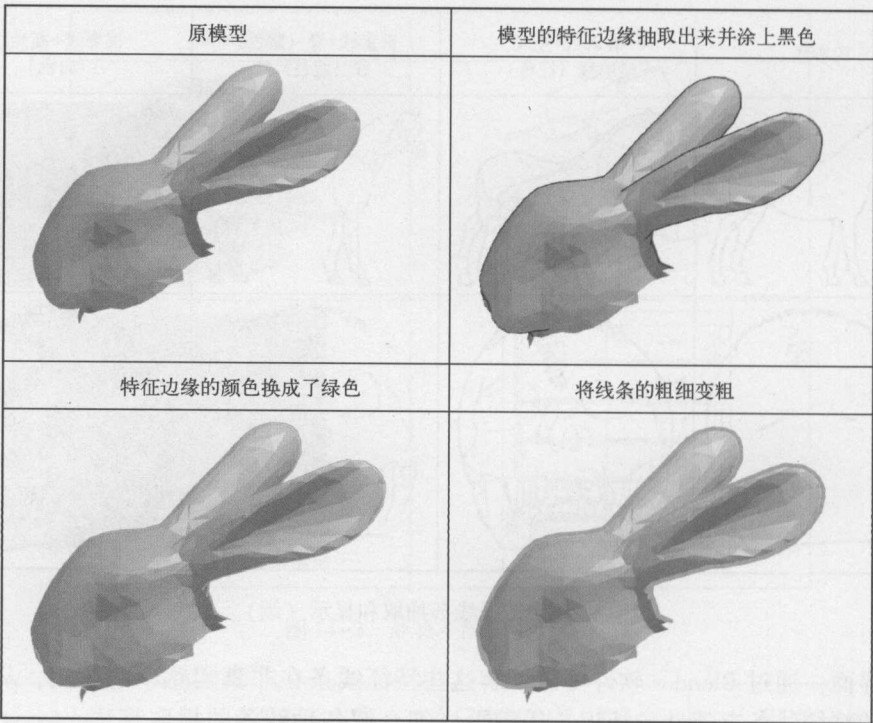


图 14-5 FreeStyle 的线条显示设置

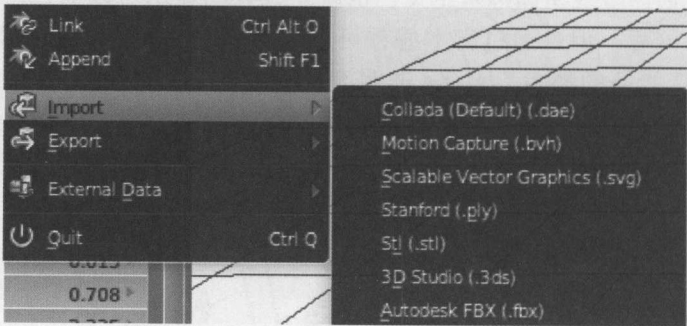


图 14-6 导入模型

S 键改变模型的大小、G 键可以移动模型、R 键可以对该模型进行旋转。通过这三个键可将模型调整成预期的样子。可以看到工作台中除了模型以外，还包括一个摄像机和光源点，如图 14-7 所示。

第三步：从图 14-8 中可以看到，渲染出的模型有些部分有阴影出现，这是因为光源点位置的关系。可以通过改变光源的位置，将其移动到模型的正前方，这样可以消除阴影。

第四步：更简单的做法是，去掉模型世界环境中的环境光遮蔽。这样渲染出的模型效果看起来更加明亮，如图 14-9 所示。

第五步：在属性窗口→渲染选项卡→freeStyle 面板上勾选复选框，这样可以开启渲染层下的 freestyle 功能，如图 14-10 所示。



图 14-7 模型调整



图 14-8 阴影调整



图 14-9 环境光遮蔽界面

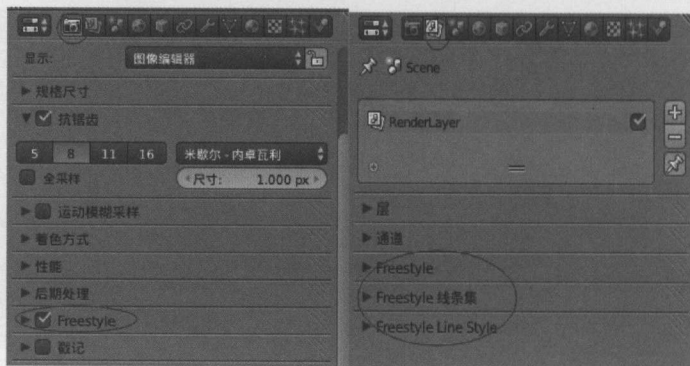


图 14-10 freestyle 选项界面

第六步：在设置了 freestyle 选项的时候，可以看到渲染出的结果在原有模型的基础上还绘制出了其基本的轮廓，如图 14-11 所示。

第七步：在 freestyle 的渲染层中，有两种控制模式，一种是参数模式，一种是 python 脚本模型，如图 14-12 所示。



图 14-11 freestyle 轮廓线



图 14-12 freestyle 模式界面

第八步：脚本模式可以进行脚本编程控制渲染的效果，freestyle 提供一些内置的脚本，利用其内置的一些脚本文件，也可以渲染出非常有趣的效果，如图 14-13 所示。

第九步：在 freestyle 线条集中，提供了几种常用的特征线条类型。选择的边类型不同，渲染出的效果也不一样，如图 14-14 所示。

第十步：freestyle 支持的各种特征线条显示效果如图 14-15 所示。

第十一步：为了更好地凸显 freestyle 的非真实感渲染效果，还有对模型的材质及贴图进行修改。材质中主要修改它的漫射及高光，将这两者的效果设置为卡通效果，并将高光的强度设置为 0。材质中，主要是改变其纹理，将纹理的类型选择为图像/影片，如图 14-16 所示。

第十二步：做完卡通渲染的设置后，再次对模型进行渲染，效果如图 14-17 所示。

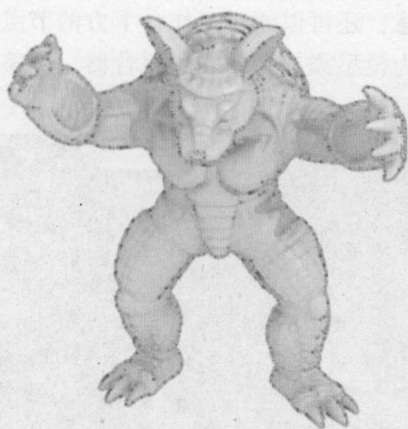


图 14-13 内置脚本渲染结果

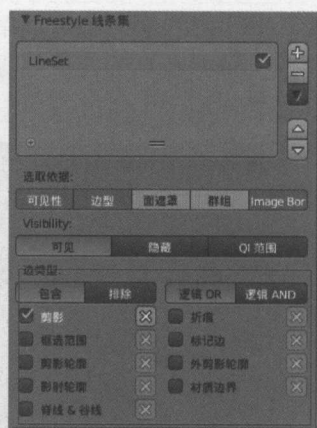


图 14-14 特征线条选择

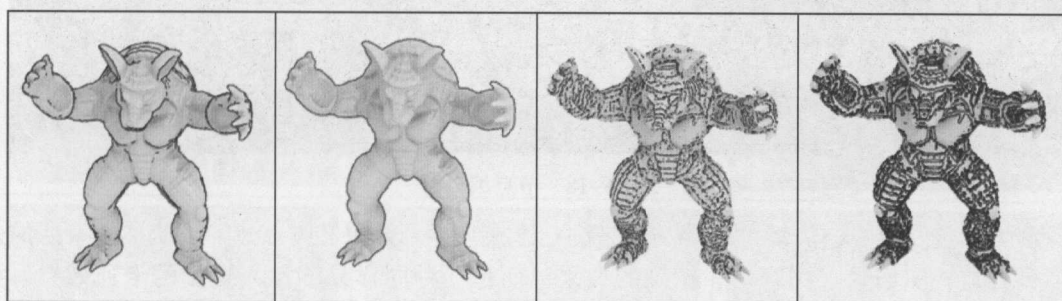


图 14-15 freestyle 支持的特征线条



图 14-16 freestyle 非真实感卡通渲染设置

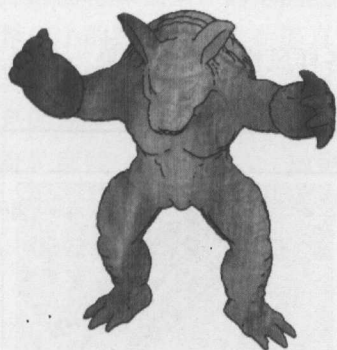


图14-17 freestyle 卡通渲染效果

第十三步：如果对模型渲染后的结果不满意，还可以通过工作台下方的节点编辑对模型的参数进行调整。选择节点的合成处理。可以为模型添加新的颜色混合器，调整其样式，改变其渲染后的结果，如图 14-18 所示。

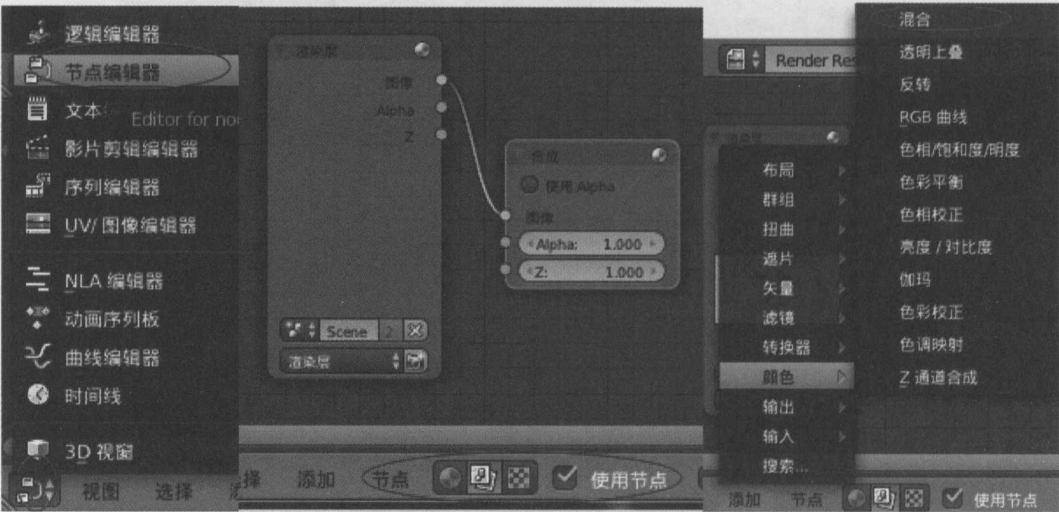


图 14-18 节点编辑器

第十四步：最终渲染后的结果与原来的三维模型显示效果相差甚远，非真实感卡通的效果不具有与现实场景一样真实、精确的效果，而是一种同样易于让人接受的卡通效果，如图 14-19 所示。

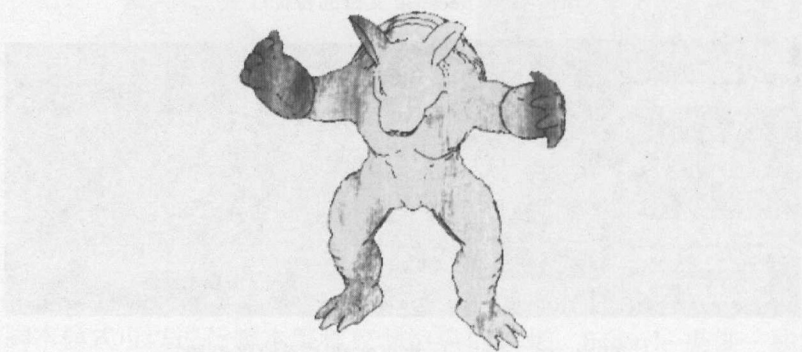


图 14-19 最终渲染效果

通过 freestyle 可以制作各种各样风格的渲染效果，各种三维模型的 freestyle 非真实感渲染效果如图 14-20 所示。

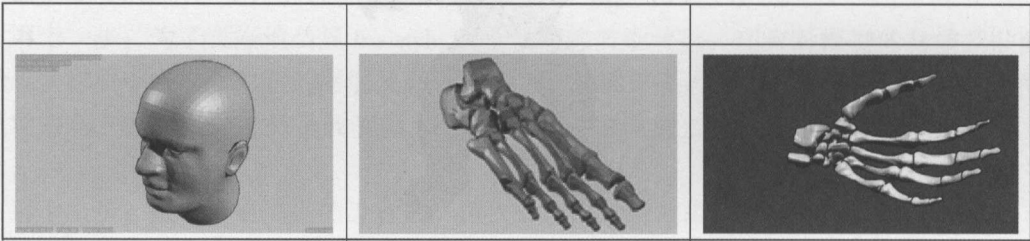


图 14-20 各种三维模型的 freestyle 非真实感渲染效果

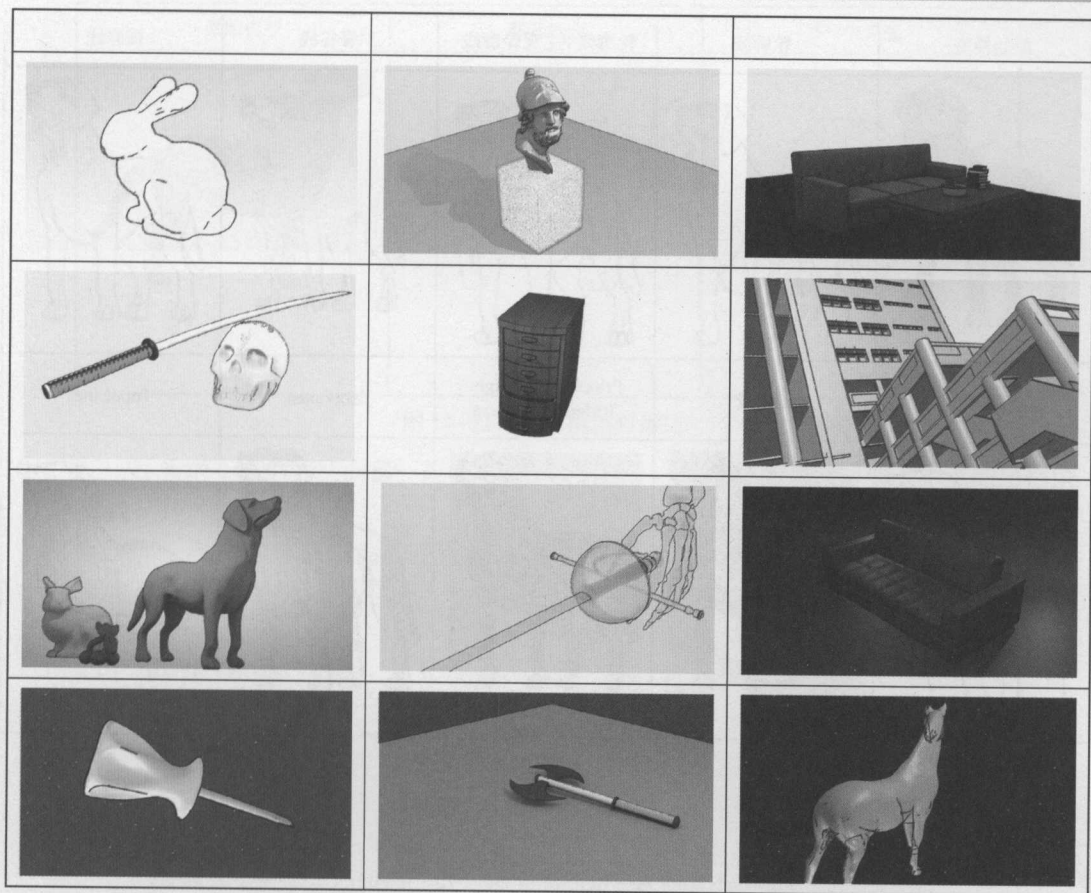


图 14-20 各种三维模型的 freestyle 非真实感渲染效果 (续)



14.2 特征线条分类

在非真实感渲染中，最重要的环节是特征线条的提取，也就是根据三维模型、当前的视角、光照等条件，计算一些具有某种特点，或者能描述三维模型特点的线条。三维模型的特征线条源于模型特征，包括模型外部的轮廓线条、内部的细节特征线条、褶皱、凹凸等。非真实感渲染可以用简单而且数据量极小的线条就可表达逼真且数据量极大的三维模型。这些线条符合视觉效果且具有说服力。除了特征线条外，三维模型表面的标记、颜色和反射率的变化这些也都是三维模型视觉信息的重要组成部分。

常用的三维模型的特征线条有轮廓线 (Contours)、剪影线 (Silhouette)、脊谷线 (Ridges & Valleys)、主观轮廓线 (Suggestive Contours)、显式脊线 (Apparent Ridges)、高光线 (Highlight Line)、分界线 (Demarcating Curves)、光极值线 (Photoc extremum lines)、拉普拉斯线 (Laplacian Lines)。

图 14-21 是这几种特征线条的图示。











原始模型	轮廓线	轮廓线+主观轮廓线	脊谷线	视脊线
				
剪影线	高光线	Principal Highlight Ridges & Valleys	Isophotes	TopoLines
				

图 14-21 三维模型特征线条



14.3 剪影线

1. 概述

剪影线（Contours）指的是三维模型的外围轮廓。它与观察模型的视点有关，当模型的视点发生了变化，模型的外围轮廓也会发生变化。剪影线是模型上最基本的特征线条。给定一个三维模型和一个视点，剪影线就是模型表面可见与不可见部分的公共边界轨迹的集合。剪影线是最简单且最容易理解的一种线条抽取算法，算法实现简单，绘制出的线条与视点的位置有关。但剪影线线条绘制出的只是模型的外围轮廓，一些模型的内部细节信息无法精确的描绘出来，需要配合其他线条才能达到最好的效果。从当前视点观察，剪影线是一个深度不连续的序列。剪影线位于三维模型三角形面法向量与视线向量垂直的地方。视线向量是从视点到模型上一点的向量，剪影线是一种视点依赖的线条。一般来说，剪影线不足以捕捉到与模型所有感知相关的特征。从图 14-22 中可以发现，三维模型很多重要的细节并没有捕捉到。因为它们只依赖与法向量的变化，所以剪影线是一阶曲线。

2. 算法步骤

第一步：计算三维模型每个面的法向量 N 。

第二步：确定当前视点的位置。

第三步：如果面的法向量与视线向量的点积为负值，则此面为反面，反之，此面为正面。

第四步：当一条边由正反两面共享时，这条边就是轮廓线。

第五步：判断剪影线的公式为

$$((v_0 - V) \cdot N_1) * ((v_0 - V) \cdot N_2) \leq 0$$

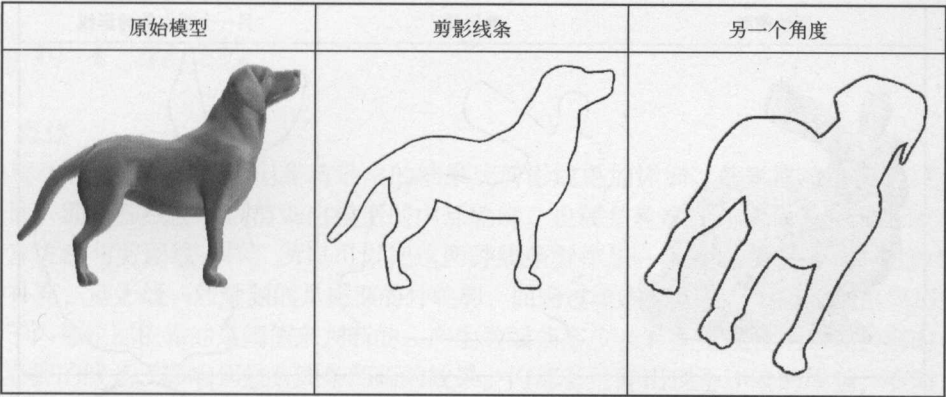


图 14-22 三维模型剪影线

其中， N_1 、 N_2 为两个面片的法向量， $v_0 - V$ 是视线向量， v_0 为两个面片共享边上中点的坐标值。

3. 代码

(1) 计算一条边相邻两个面的正反值。

```
public double ComputeValue(TriMesh. Edge edge)
{
    double value1, value2;
    //考虑到边界的情况
    if (edge.Face0 == null || edge.Face1 == )
    {
        return 0;
    }
    //正常情况
    Vector3D normal = TriMeshUtil. ComputeNormalFace(edge.Face0);
    Vector3D dir = edge.Vertex0. Traits. Position - viewPoint;
    value1 = dir. Dot(normal);
    normal = TriMeshUtil. ComputeNormalFace(edge.Face1);
    value2 = dir. Dot(normal);
    return value1 * value2;
}
```

(2) 遍历所有面，查找正反值不一样的边。

```
public override TriMesh Extract() {
    for (int i = 0; i < mesh. Edges. Count; i++)
    {
        if (LineCurvature. ComputeValue(viewPoint, mesh. Edges[i])
            < ConfigNPR. Instance. Contourthreshold)
        {
            mesh. Edges[i]. Traits. selectedFlag = 1;
        }
    }
    return mesh;
}
```

4. 效果图

图 14-23 展示了各种不同形状和拓扑的三维模型的剪影线图示，因为剪影线条随着观察角度的不同而不同，第三列展示的是不同角度的剪影线条对比。

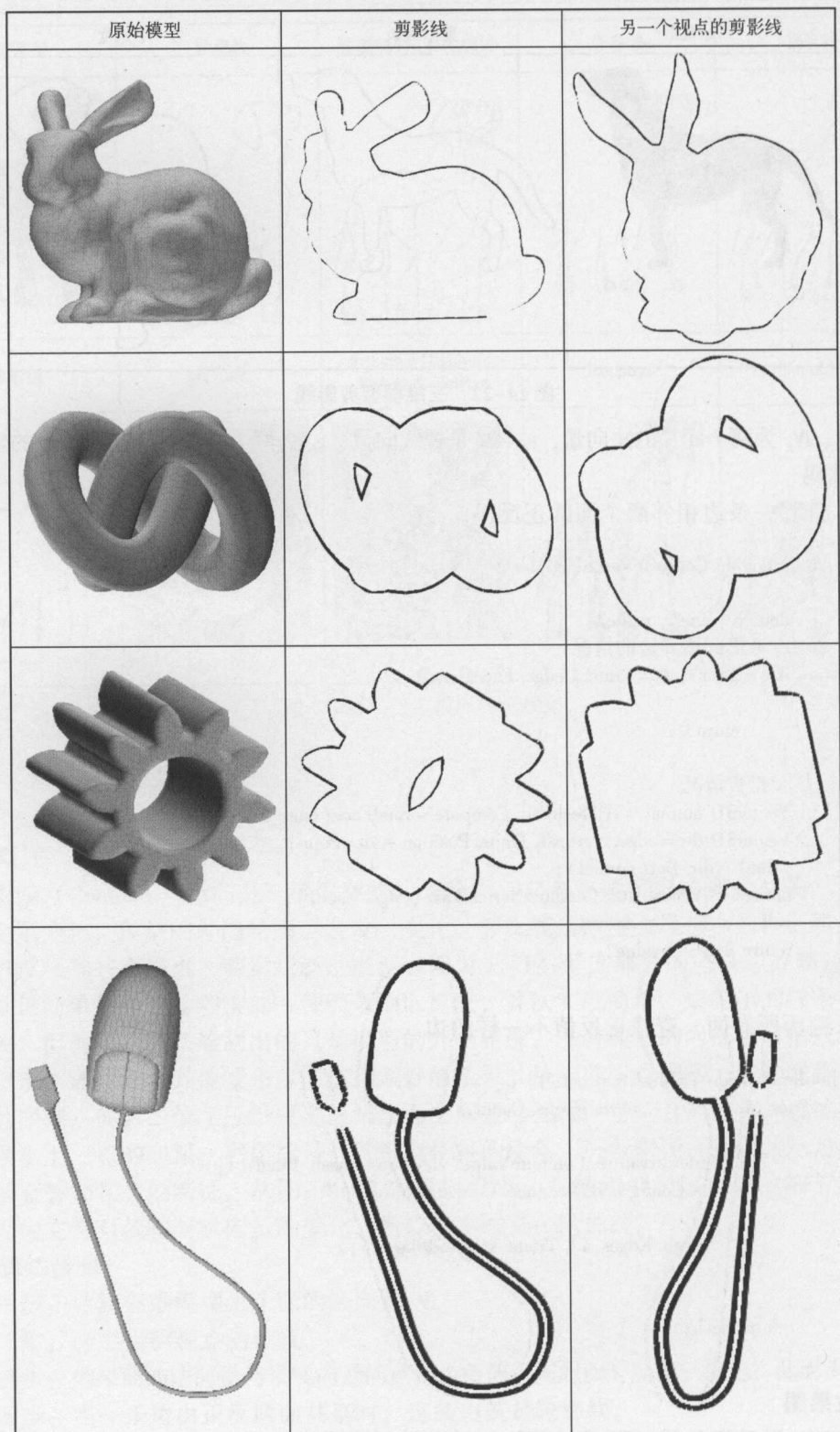


图 14-23 三维模型不同角度剪影线



14.4 轮廓线

1. 概述

轮廓线和剪影线类似，但是渲染出的结果比剪影线更加详细。剪影线渲染出的只是模型的外围轮廓，而轮廓线还可以渲染出模型的内部轮廓。也就是轮廓线包含内轮廓和外轮廓。轮廓线的计算方法与剪影线一样，所以可以把这两种线条看作是一种线条。轮廓线是对剪影线线条的一种补充，剪影线一般绘制的是模型的外轮廓，而通过轮廓线算法还可以绘制出模型的内部轮廓。由于内部轮廓依靠的是阈值来判断的，产生的线条多少会有些生硬且不自然，有时也会产生一些不必要的线条，影响到整个线条图画的效果。内部有可能出现不连续的线条，有时可能要设置多个阈值来达到更好的效果。对于不同的模型，阈值设置可能不一样，如图 14-24 所示。

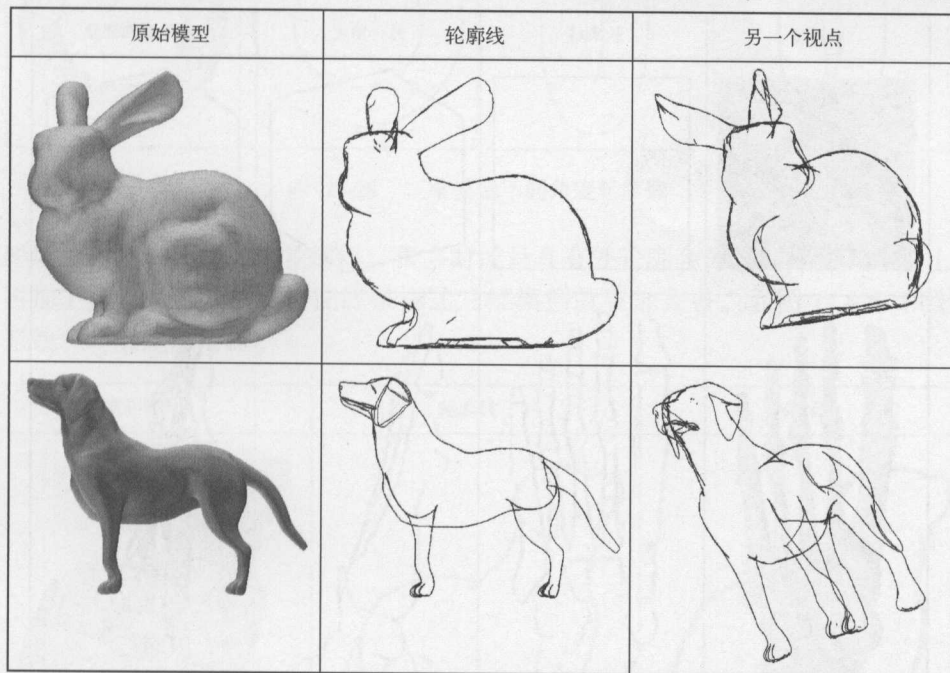


图 14-24 三维模型轮廓线

2. 算法步骤

第一步：剪影线条的提取，主要是确定模型的内外轮廓，如图 14-25 所示。

第二步：外轮廓是指相对于当前观察者的方向而言，面向不同朝向的两个公共边界，也就是轮廓线。

第三步：内轮廓是两个多边形共有的边界，两个面的夹角小于一个阈值。所以，要为内轮廓线指定一个阈值。

第四步：阈值确定后，计算视线向量与模型各个面的法向量夹角，得到外轮廓。

第五步：计算两个相邻面法向量的夹角值，与阈值进行比较，确定内轮廓。

第六步：如图 14-26 所示，面 A 和面 B 在交界处（红点）形成内轮廓，在该交界处，面 A 的发现和面 B 发现夹角大于 90° ；视点 和面 B 所有的点的法线夹角都大于 90° ，除与面 A 和面 C 的交界处，B 面上不形成轮廓。面 C 和面 B 在交界处（蓝点）形成外轮廓，面 C 与实现的夹角小于 90° 。

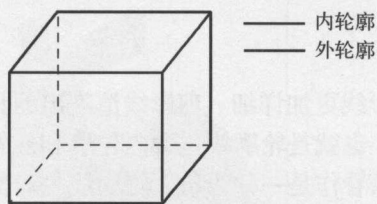


图 14-25 剪影线条的提取

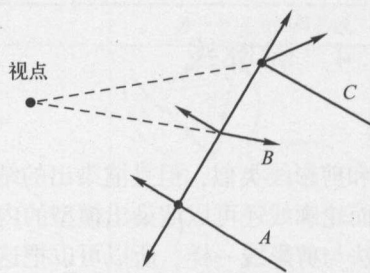


图 14-26 内轮廓的形成

3. 效果图

图 14-27 和图 14-28 展示的是各种不同形状的三维模型在两个不同角度的轮廓线，包括内轮廓和外轮廓。从中可以看出加上内轮廓线后，更能够描述三维模型的形状。

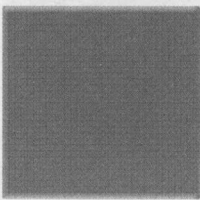
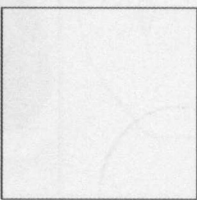
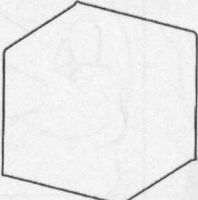
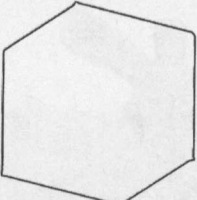
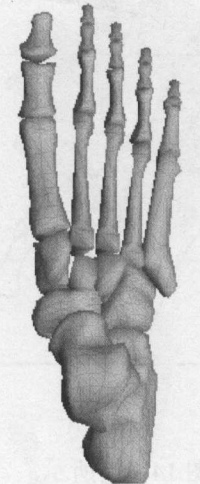

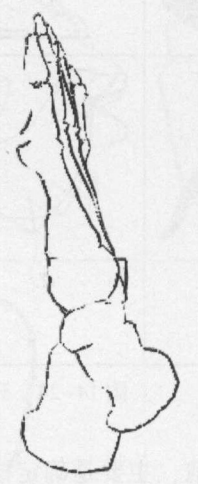
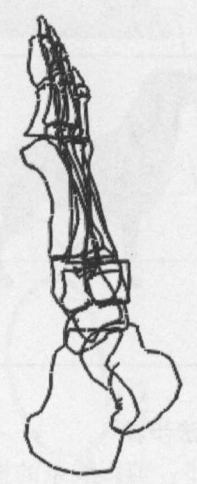
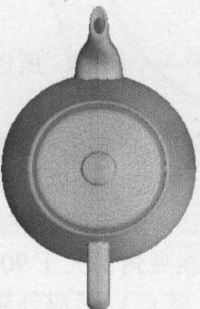
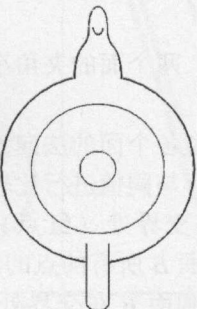
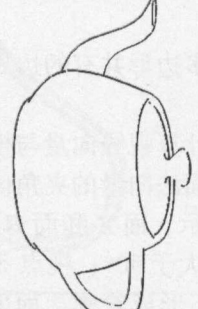
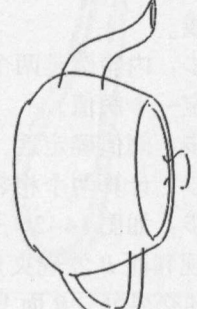
原模型	轮廓线	另一角度	显示遮挡部分
			
			
			

图 14-27 三维模型不同角度轮廓线

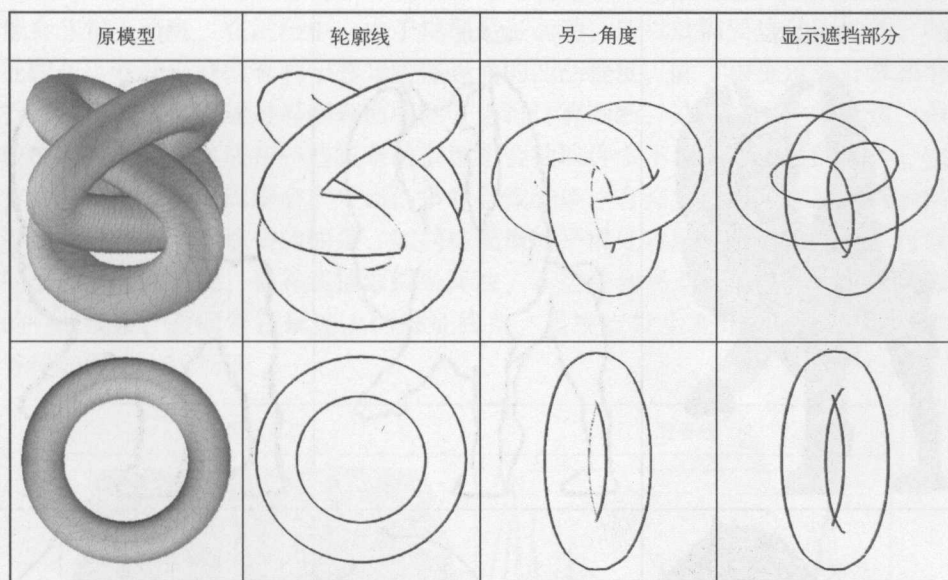


图 14-28 三维模型不同角度轮廓线

剪影线就是三维模型的外轮廓线，很多时候只具有外轮廓会失去三维模型形状上内部的细节，再加上内轮廓线后，用特征线条描述三维模型就更加完备。如图 14-29 和图 14-30 所示剪影线和轮廓线的对比。

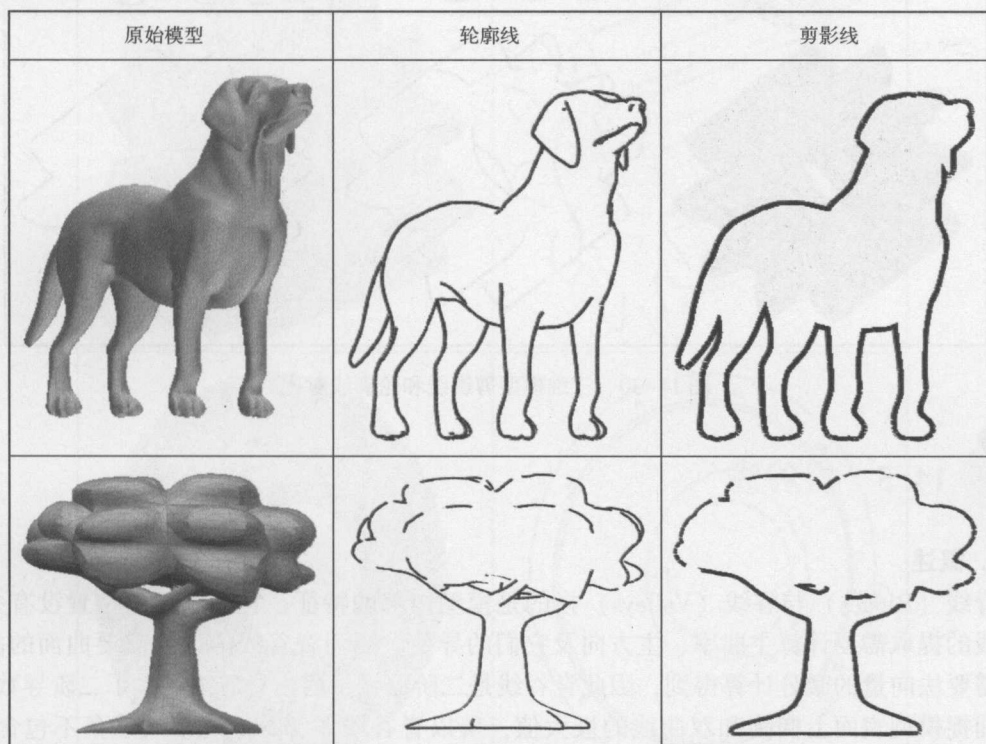


图 14-29 三维模型剪影线和轮廓线对比

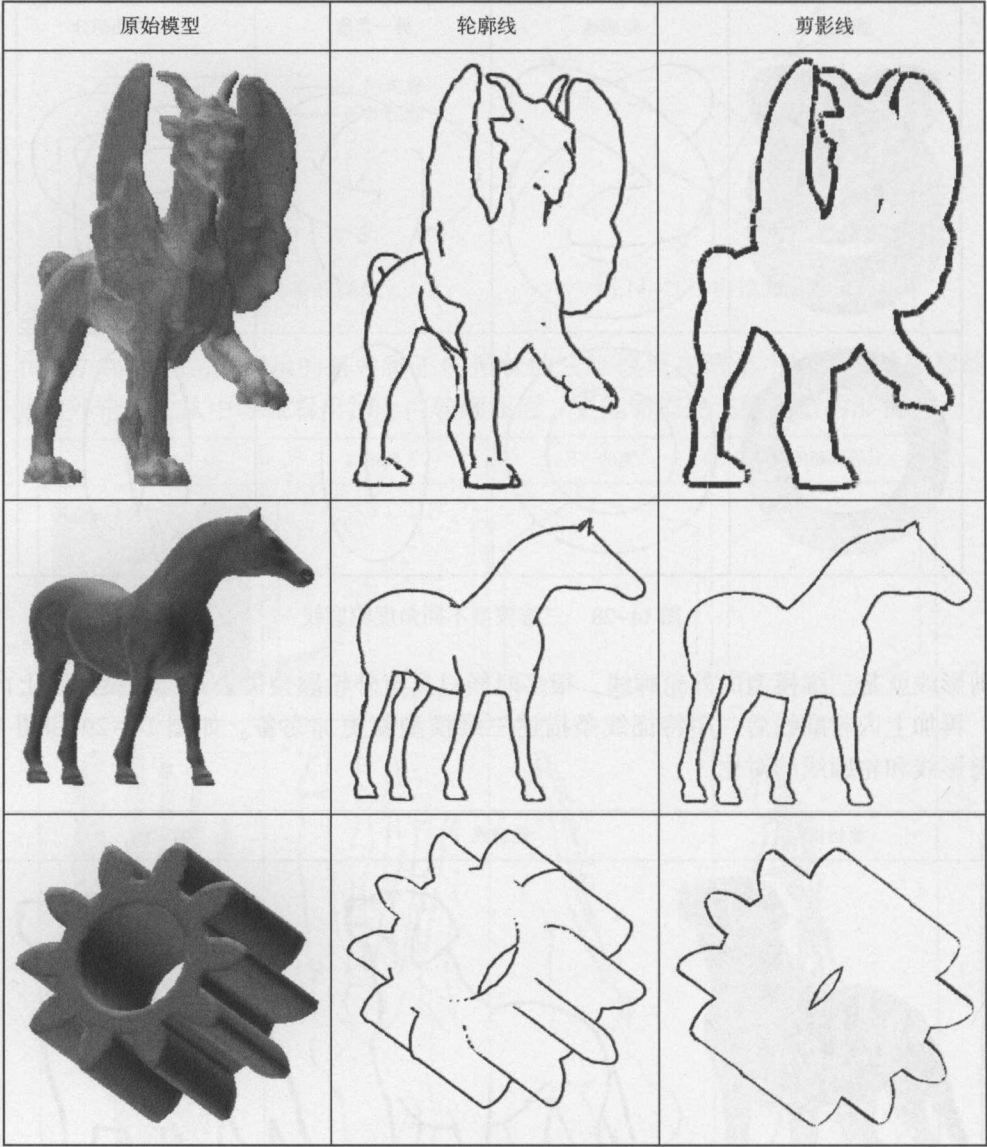


图 14-30 三维模型剪影线和轮廓线对比



14.5 脊谷线

1. 概述

脊线（Ridges）与谷线（Valleys）指的是模型内部的特征，它与视点的位置没有关系。脊谷线的提取需要计算主曲率、主方向及它们的导数。因为脊谷线的计算需要曲面的曲率，曲率需要法向量的微分计算得到，因此脊谷线是二阶曲线。因为脊谷线加入了二阶导数的计算来捕捉模型表面上椭圆和双曲线的极大值，所以脊谷线能够补充轮廓线线条不包含的信息。脊谷线的定义只考虑三维模型本身，不考虑视点的位置。脊谷线线条的缺点是，总会呈现出一些特别鲜明的折角，即使模型非常光滑。而且有些模型有很多的谷线与脊线，这样产

生的图像会非常不清晰。在动画中,由于模型总是运动,所以需要这种刚性的、与视点无关的特征线条。谷线与脊线有利于感知物体表面的凹凸程度,除了根据定义计算相邻面的二面角进行判断外,也可以通过局部表面的微分进行计算判断,这种方法比较麻烦,但是计算结果比较准确。脊谷线算法在一些复杂的模型中会抽取许多不具有连接性的线条。在模型凸起的地方一个细小的折痕周围会产生出许多脊谷线线条。有许多方法可以用来减少网格模型上这些不连续且具有干扰性质的线条,如网格模型的平滑处理或者之后阈值法,可以减少这种情况的出现。总的来说,脊谷线抽取线条算法,不会受到视点位置的影响,它抽取的是模型内部的一些线条,是固定在模型上的特征线条,当视点发生变化时,不需要重新计算。脊谷线如图 14-31 所示。

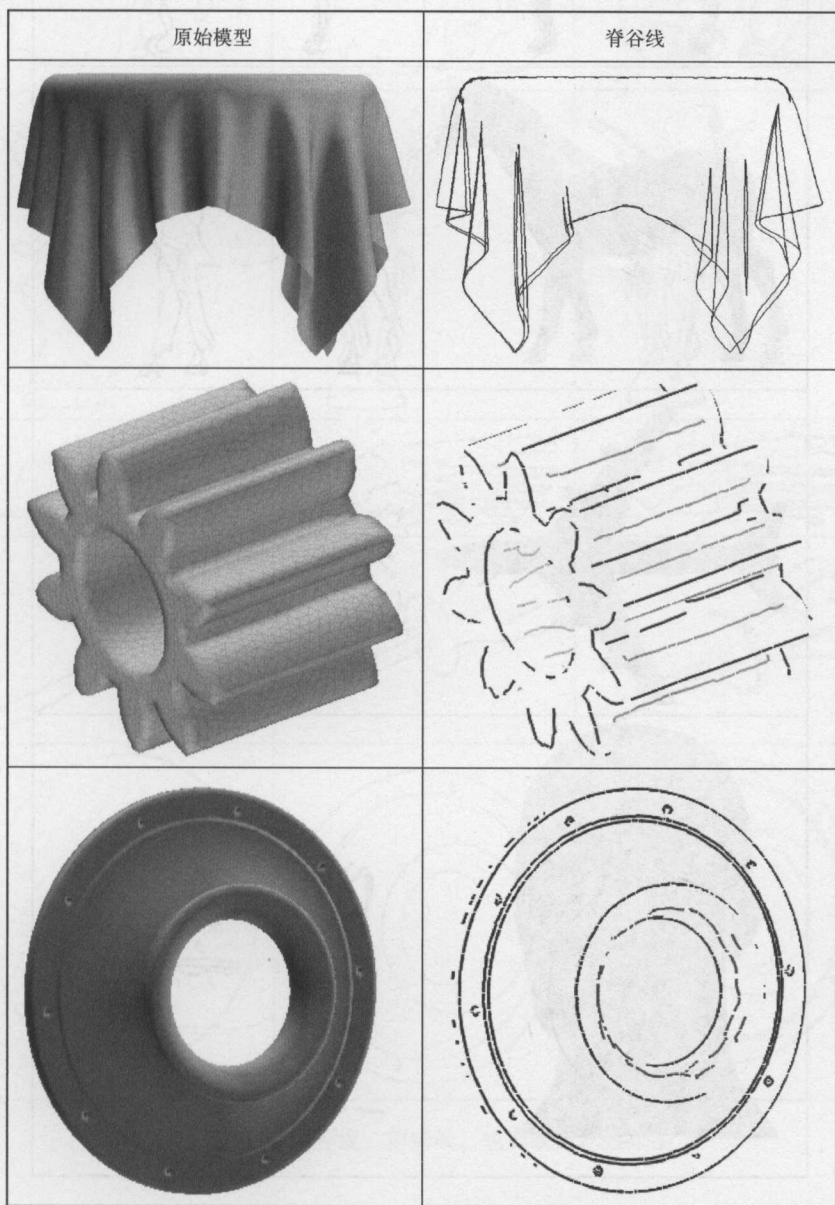


图 14-31 三维模型脊谷线

2. 效果图

图 14-32 是各种不同模型的脊谷线，从中可以看出脊谷线可以突出三维模型表面比较

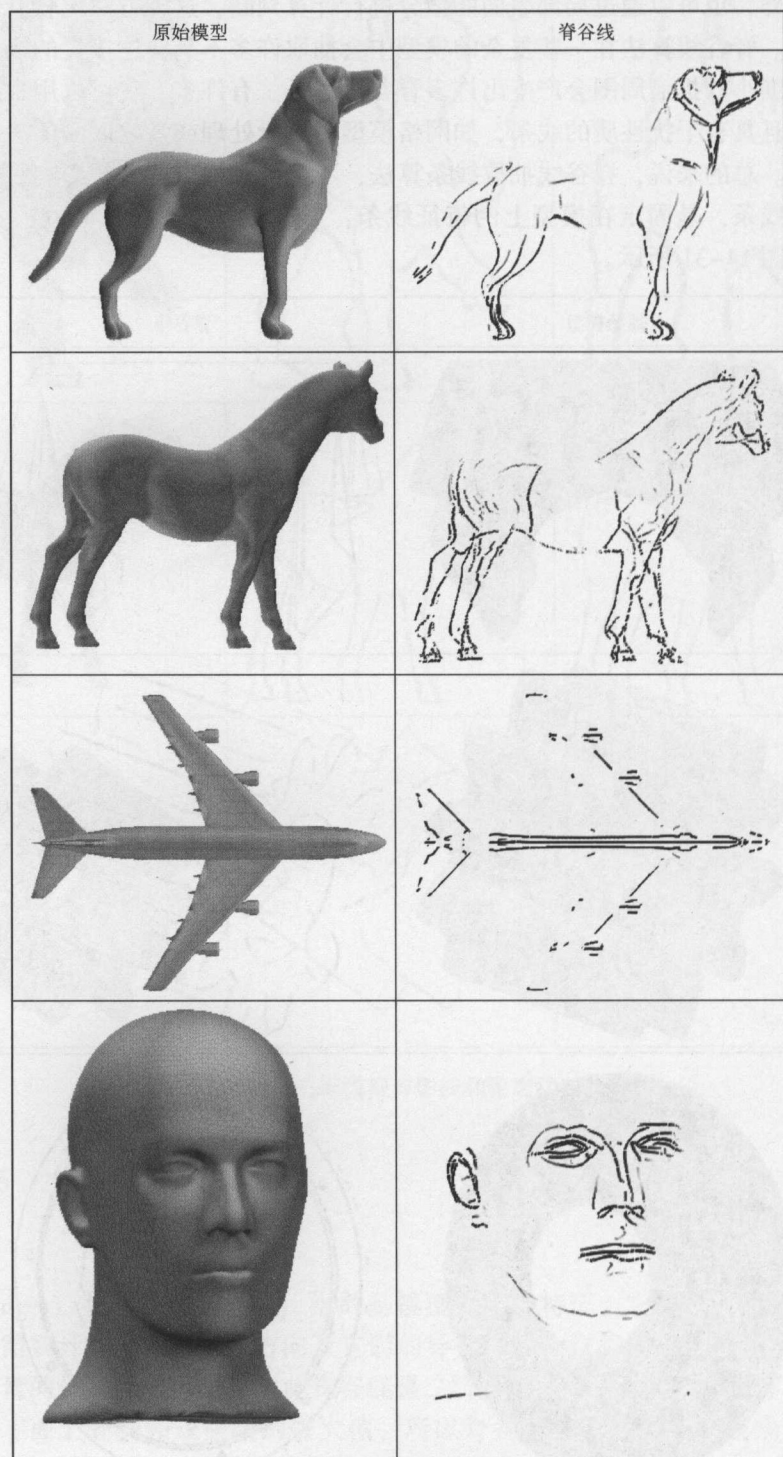


图 14-32 三维模型脊谷线效果图

凹凸的地方，但是脊谷线单独无法很完整地展示三维模型的整体形状。

通过脊谷线和其他线条的对比，可以更加清楚地了解脊谷线的特点和产生的部位，图14-33和图14-34 脊谷线、剪影线、轮廓线等的对比，通过下面的图示可以看出在轮廓线的基础上加上脊谷线之后，三维模型可以用线条显示地更加完善。

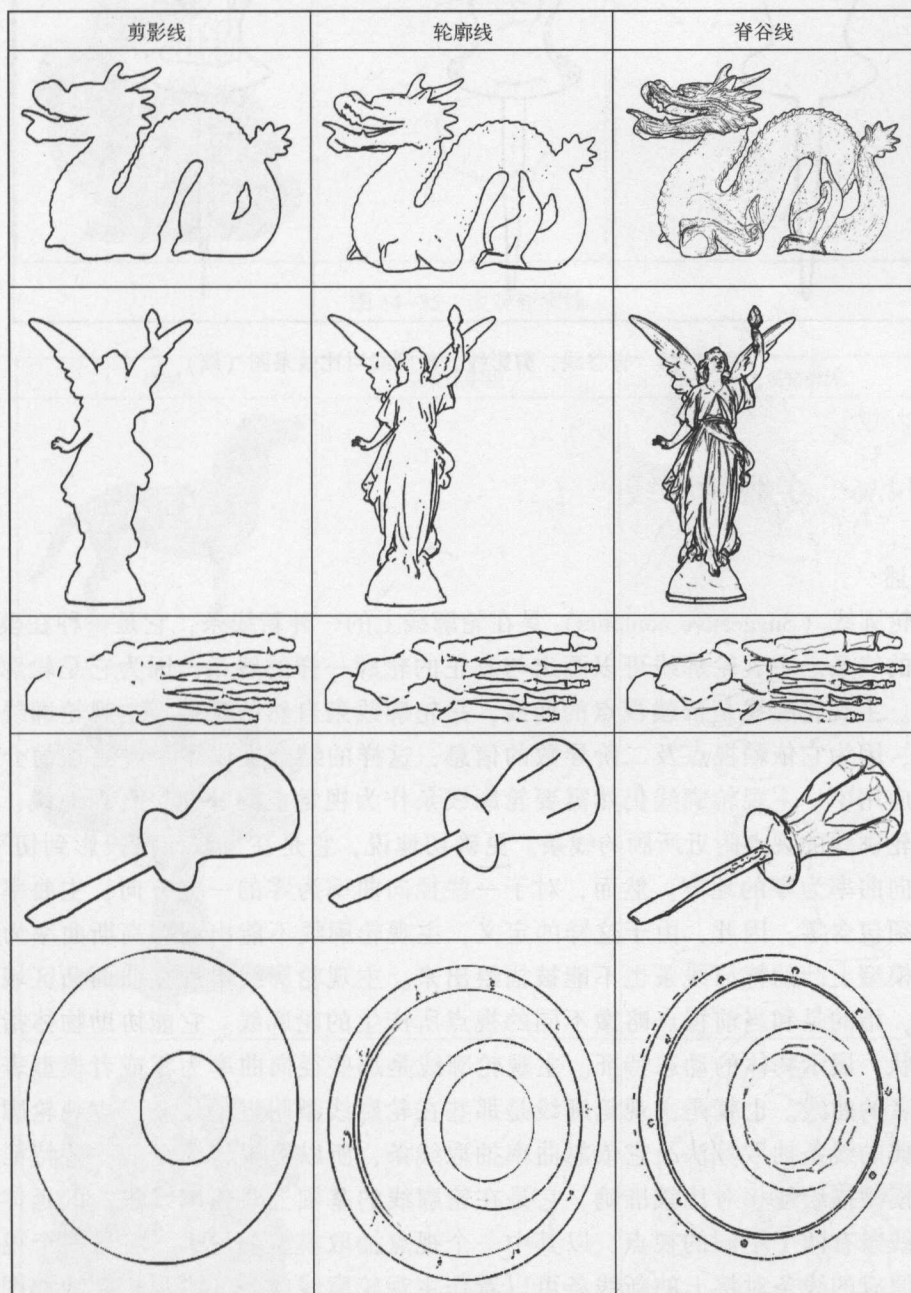


图 14-33 脊谷线，剪影线，轮廓线对比效果图

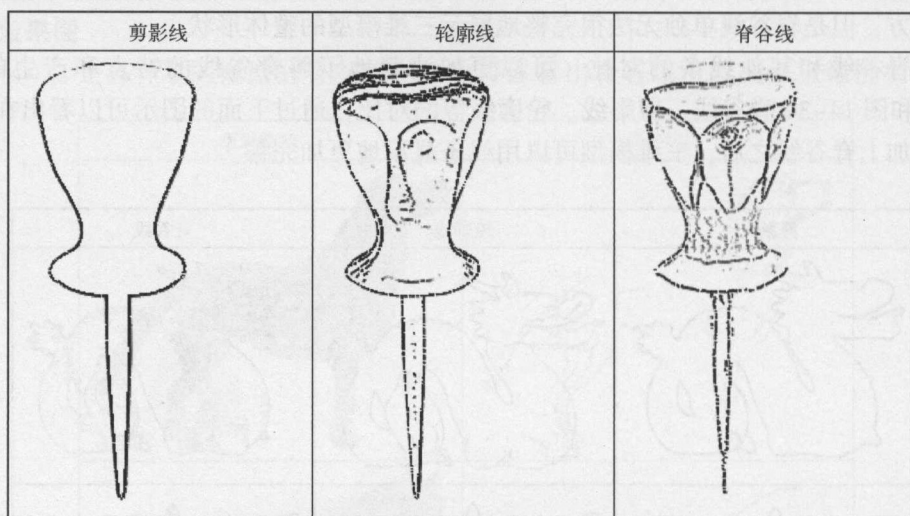


图 14-34 脊谷线，剪影线，轮廓线对比效果图（续）



14.6 主观轮廓线

1. 概述

主观轮廓线（Suggestive contours）是在轮廓线上的一种新线条，它是一种在模型表面清晰可见的线条。主观轮廓线可以画出与真正的轮廓一样的线条，因为它是轮廓线的拓展和延伸。主观轮廓线是依赖视点的曲线，是轮廓线条自然的延伸。主观轮廓线更加直观、漂亮，因为它依赖视点及二阶导数的信息，这样的结合提供了一种更加简介的线条画。实际应用中，主观轮廓线仍然需要轮廓线条作为视觉上的补充。直观地说，主观轮廓线是在轮廓线的视点附近所画的线条。更确切地说，它是在视线向量投影到切平面上，方向的径向曲率为零的地方。然而，对于一些径向曲率为零的一些方向，主曲率之间的间隔也必须包含零。因此，由于这样的定义，主观轮廓线不能出现在高斯曲率为正的地方，而且模型上凸的特征线条也不能被描绘出来。主观轮廓线出现在曲面凸区域中的可见部分中，指的是和当前视点略微不同的视点所产生的轮廓线。它能协助物体指示物体的实际形状，展示物体的动态特征。主观轮廓线是那些径向曲率为零或者模型表面弯曲远离观察者的曲线。也就是主观轮廓线是那些在轮廓线条附近的线条。主观轮廓线是一种非常经典的线条抽取算法，它依靠曲率抽取线条，所以可以将模型的一些特征线条抽取出来，模型描绘地相对比较准确，它是在轮廓线的基础上对轮廓线进行的延伸。相当于轮廓线条有两个不同的视点，以其中一个视点抽取模型的轮廓，将另一个视点抽取出的能与原有的线条对接上的新线条可以看作主观轮廓线条。主观轮廓线如图 14-35 所示。

2. 效果图

如图 14-36 和图 14-37 所示为不同模型的主观轮廓线，但是只是主观轮廓线，本身无

法完整地显示三维模型的形状，在加上轮廓线之后，主观轮廓线就能够更清楚地表示三维模型表面的某些特征。从而使三维模型的用线条表现出来的细节更加丰富。

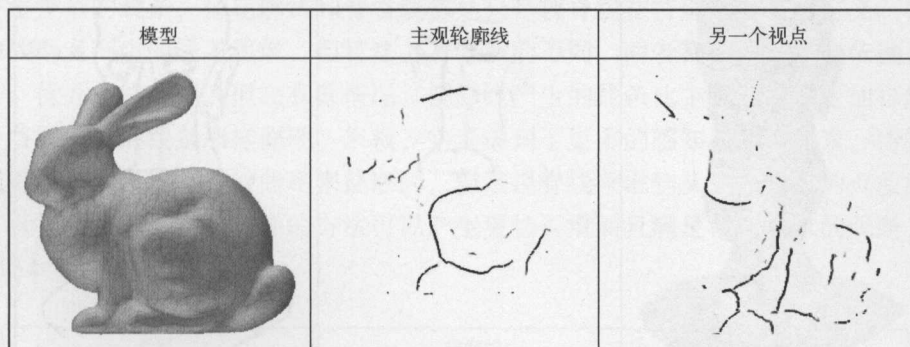


图 14-35 主观轮廓线

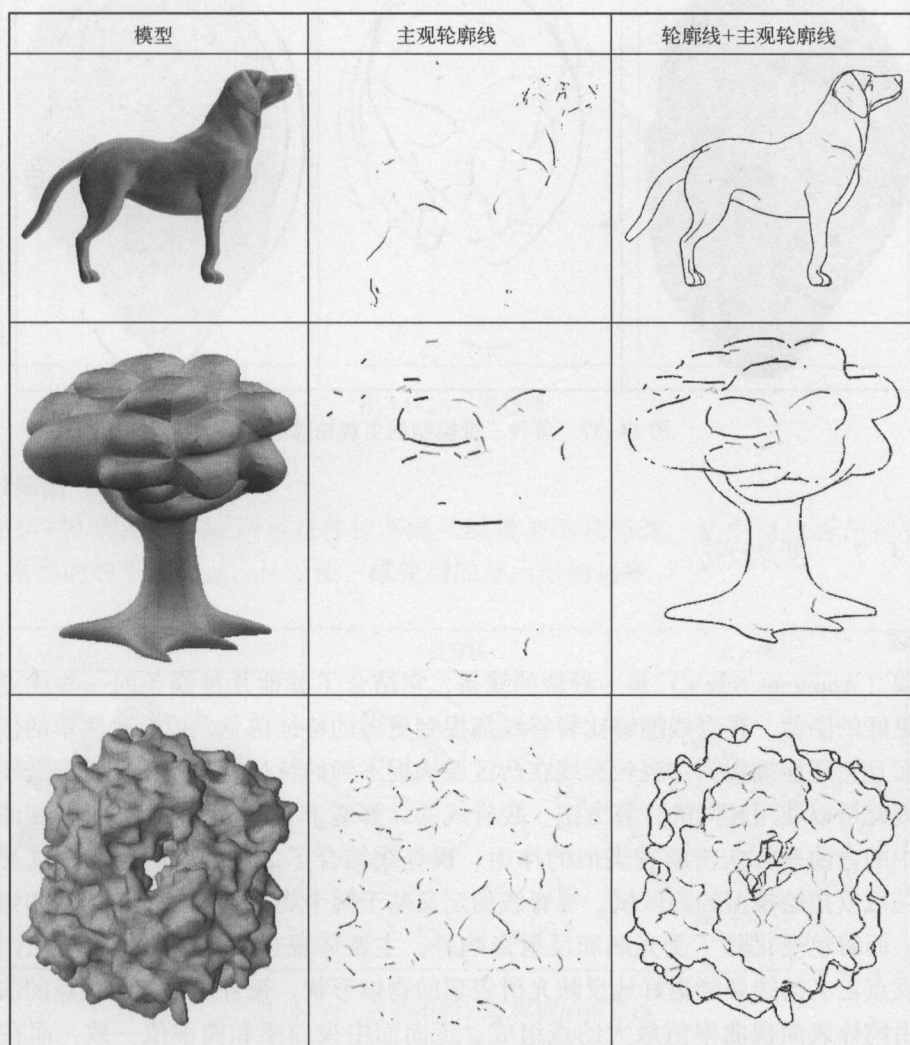


图 14-36 各种三维模型的主观轮廓线

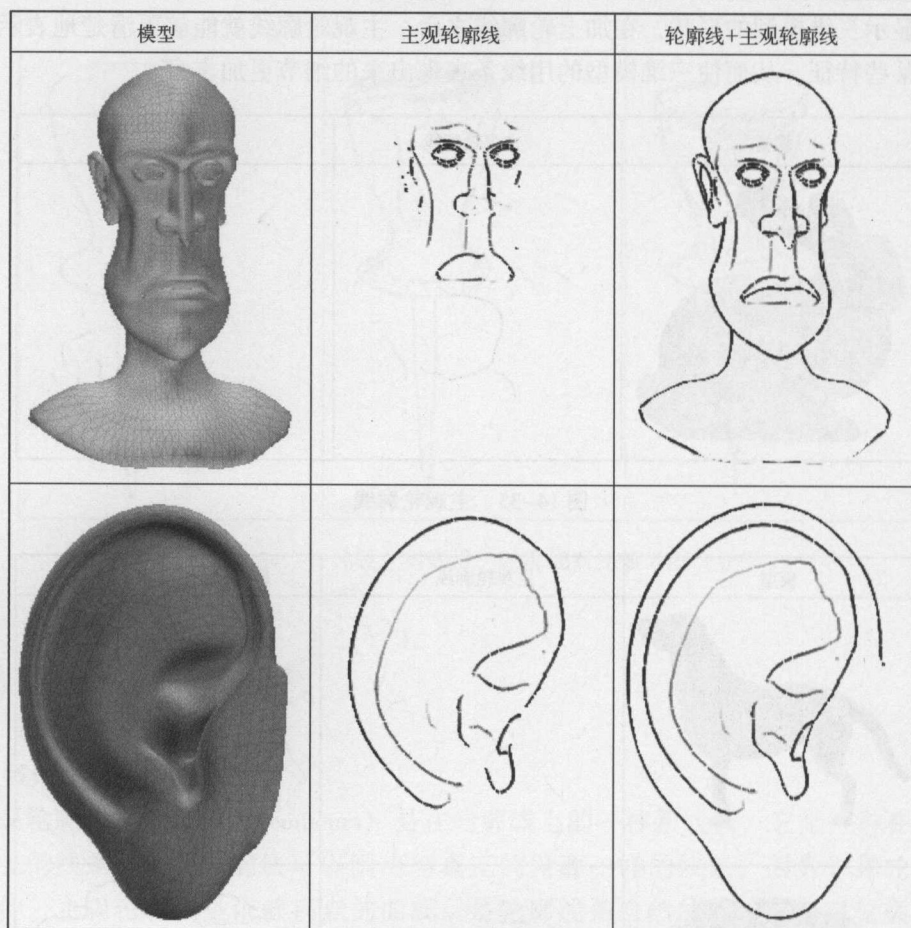


图 14-37 各种三维模型的主观轮廓线



14.7 视脊线

1. 概述

视脊线（Apparent ridges）是一种新的线条，它结合了其他几种线条的一些优点，能够产生效果更好的图像。视脊线能够比脊谷线捕捉到更多的特征信息，以一种简单的依赖视点的方式。而且它还能捕捉到主观轮廓线在凸区域捕捉不到的特征。轮廓线是视脊线线条的一种特例，因此可以使用相同的计算方法。视脊线的计算基于一种新的几何属性：视曲率，它与脊谷线中的主曲率方向扮演着类似的作用。视脊线结合了二阶导数的信息，以及依赖视点，而且还可以描绘模型的凸区域。视脊线的定义基于两个观察：一方面，人的知觉对着色变化敏感，而着色变化除了受光照和反射影响外，主要体现在曲面法向量的变化；另一方面，基于视点定义线条能更好地反映光滑表现的物体形状。视脊线是基于视点的曲率进行定义的，由物体表面视曲率值最大的点组成。正向面中视曲率和曲率值一致，而在其他视角，视曲率值将大于曲率，而视曲率能更好地反映视觉对曲率变化的感知。

视脊线描绘模型的结果与主观轮廓线很相似，但视脊线描绘的线条比较光滑，它利用视曲率计算模型的线条。视脊线能产生视觉上赏心悦目的特征线条，可以捕捉到模型上重要的信息。许多类型的线条，如轮廓线和脊谷线都是只是视脊线在特殊情况下的定义。视脊线与主观轮廓线两者产生的线条很像，但算法上有本质的不同。两者都是基于视点依赖，都可以产生简洁、优秀的线条图。但在有些情况，视脊线产生的线条比主观轮廓线更加自然。视脊线不像脊谷线产生的线条那样僵硬、呆板，它考虑到了更多的感知相关的因素。传统的线条与视脊线都是基于模型上点的曲率来抽取的，但是视脊线考虑到从一个特定的点使用物体的透视图来计算视曲率。通过这样的方法可以产生更加光滑而且满足视觉需求的线条。

如图 14-38 所示为视脊线。

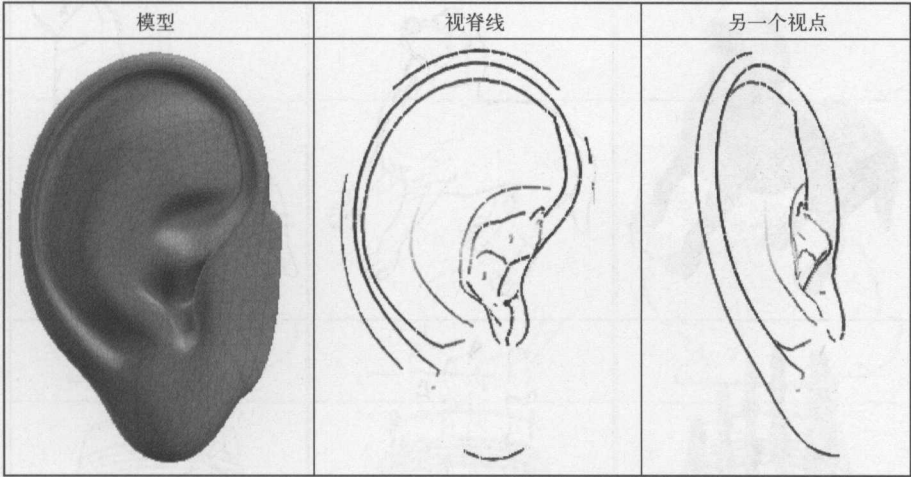


图 14-38 视脊线

2. 效果图

如图 14-39 和图 14-40 所示是各种不同三维模型的视脊线，从中可以看出视脊线可以表示三维模型的细节和特点，从而使三维模型的显示更加清晰。

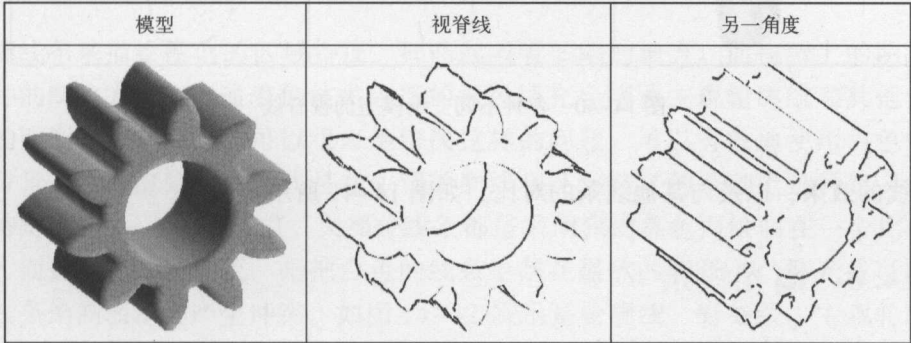


图 14-39 各种不同三维模型的视脊线

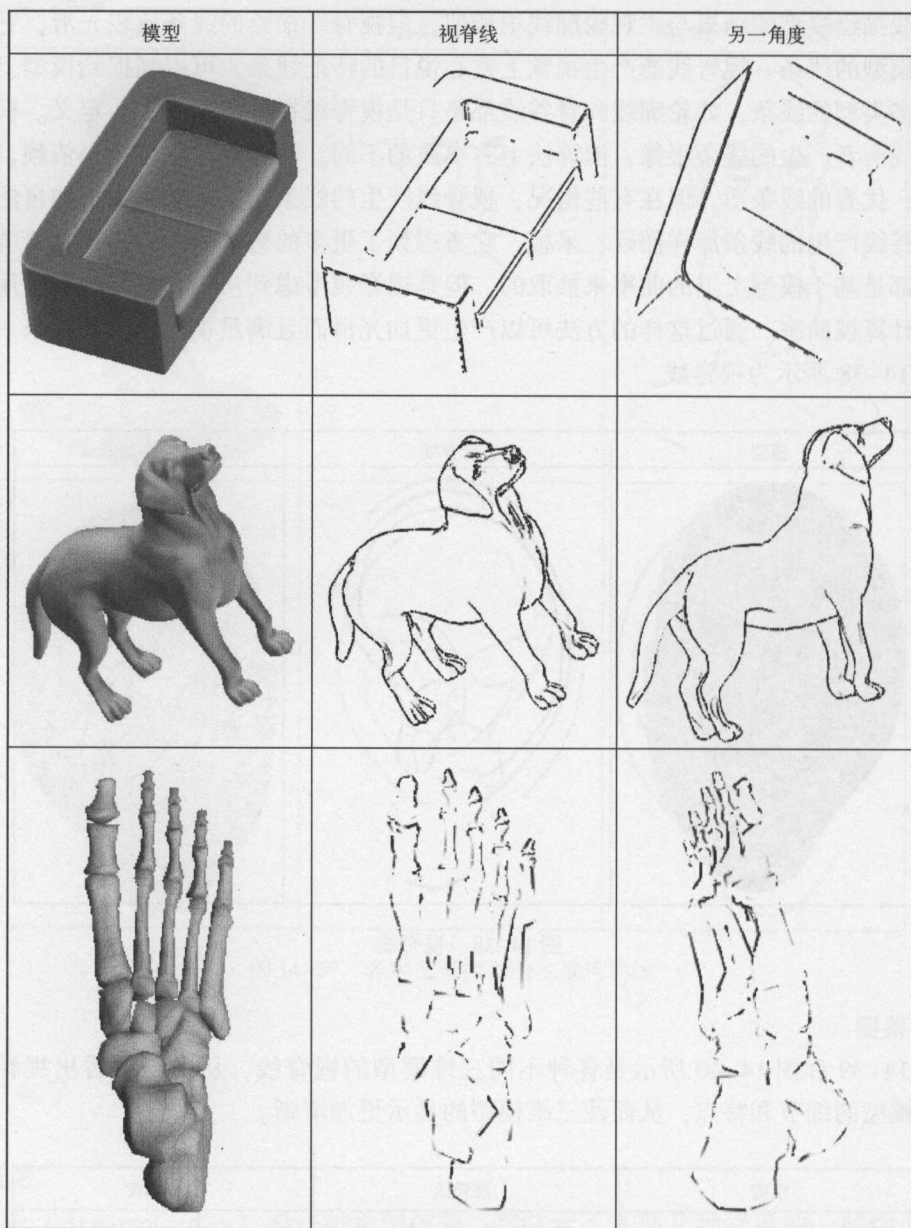


图 14-40 各种不同三维模型的视脊线

视脊线的效果，以及与其他线条的对比，如图 14-41 所示。



14.8 高光线条

高光线条（Highlight Lines）建立在主观轮廓线的基础上，主要有两类线条：依靠视点的变化而变化的主观高光线（Suggestive Highlights）和由模型表面的主曲率方向决定的主曲率高光线（Principal HighLights）。在某些三维模型形状的椭圆形区域，无法使用轮廓线和

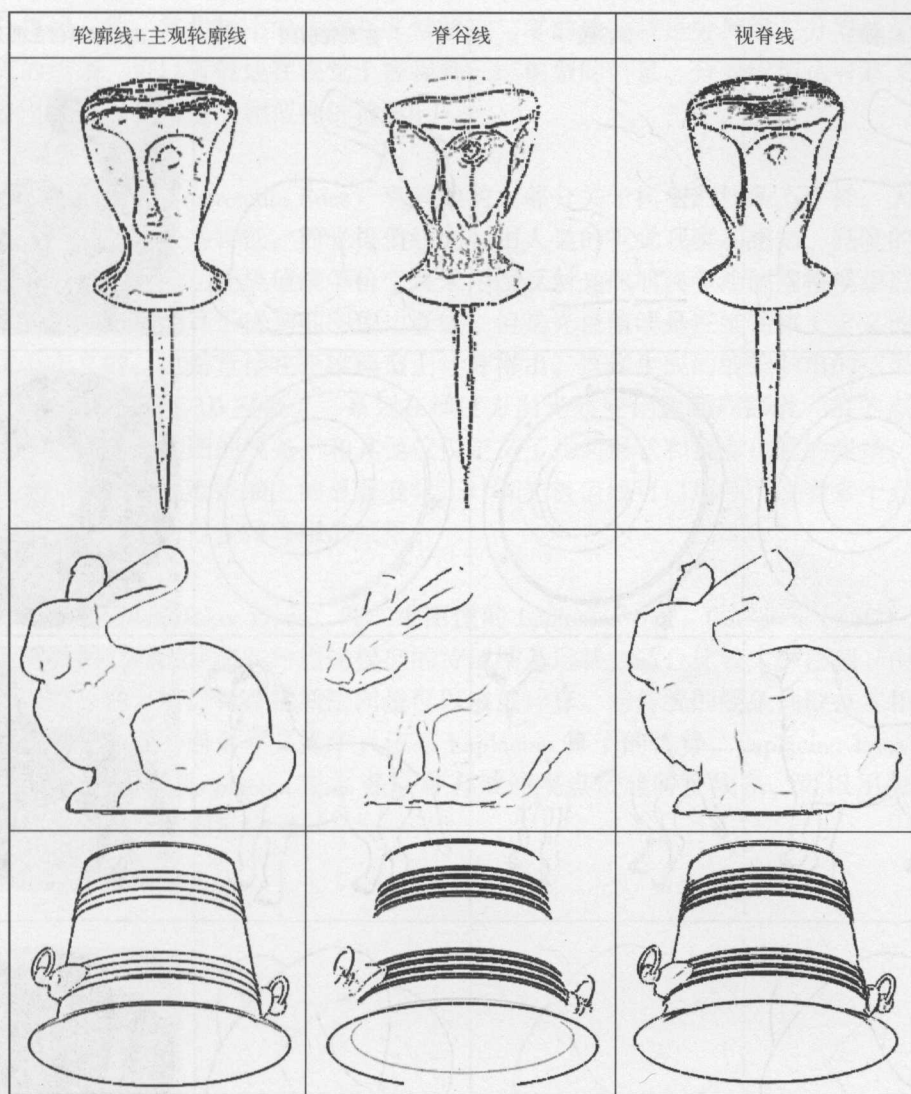


图 14-41 主观轮廓线、视脊线、脊谷线对比

主观轮廓线条来描绘模型的区域特征，如凸起或者凹陷的地方。而模型上的阴影提供了一个很好的解决方案，卡通着色技术创造的深色填充会掩盖主观轮廓线和其他深色线条在模型的阴影区域。高光线可以很好地解决这样的问题，可以有效地使用白色的线条在深色的背景上进行渲染。高光线是在主观轮廓线和几何褶皱的基础上进行定义的。高光线是特定方向 $v \cdot n$ 的最大值。大部分线条都是采用深的颜色而绘制在一个比较明亮的背景上，而高光线正好相反。是把白色的线条绘制在黑色的背景上。高光线可以配合其他各种线条绘制而不会产生冲突。如图 14-42 所示是轮廓线、脊谷线、主观轮廓线和高光线的对比。

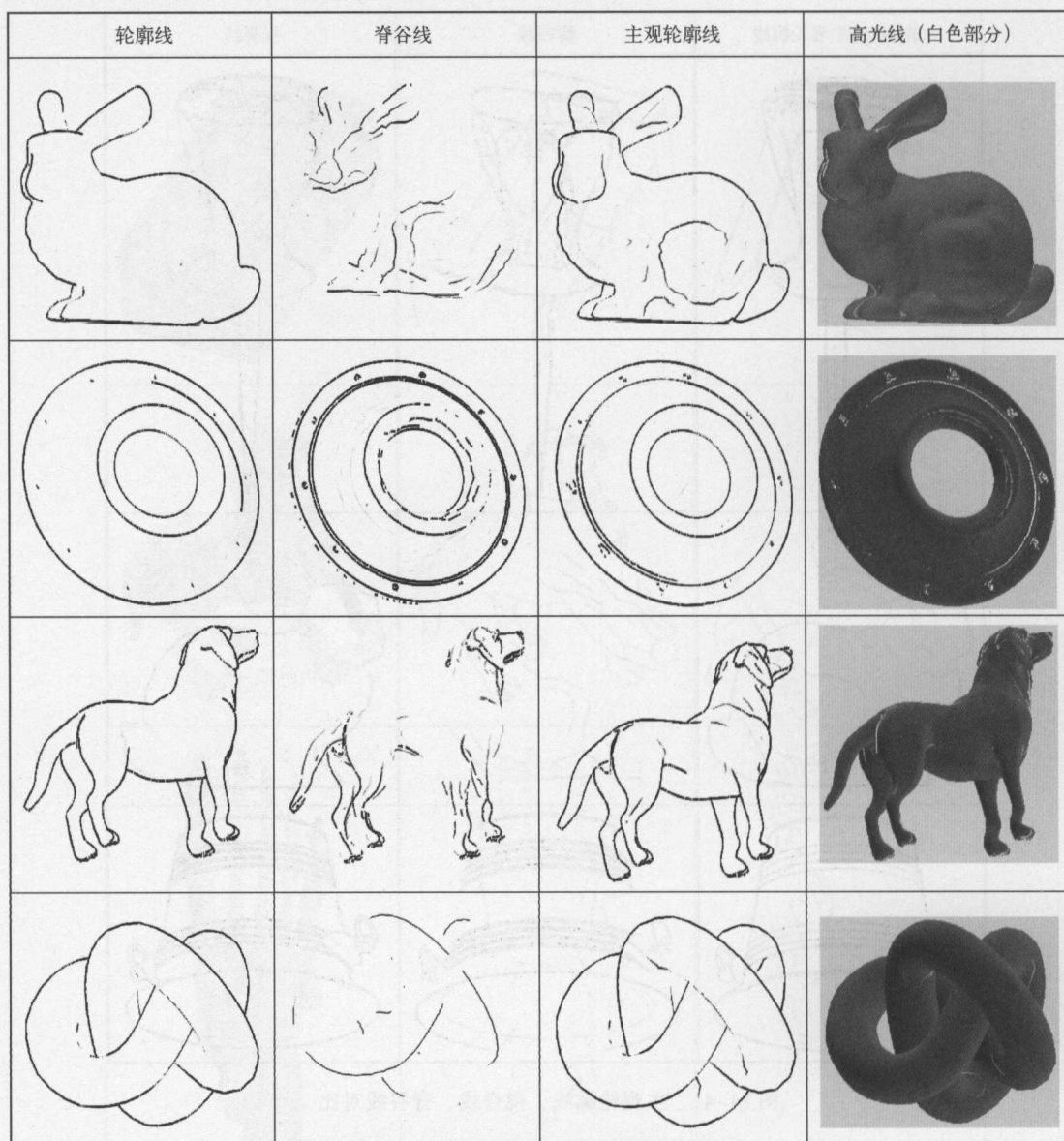


图 14-42 轮廓线、脊谷线、主观轮廓线和高光线的对比



14.9 其他线条

1. 分界线

分界线（Demarcating Curves）与视点的位置无关，可以传达固有的特征形状，强调 3D 纹理。分界线类似于某些技术绘图的线条，如考古文物的草图。分界线适合考古碎片和文物模型等三维模型上线条的绘制。分界线是在曲面上固定且剧烈弯曲处的特征线条，与脊谷线一样，固定在模型的表面上，不会因为视点的变化而变化。分界线不依赖于视点并出现在模型弯曲处的线条的存在，这样的地方的高斯曲率为零或者平均曲率为零。一般情况下，这些

曲线与分界线一致。其中较小的曲率是非常接近与零的地方就是分界线。分界线是一种不依赖于视点的线条，可以有效地在视觉上捕捉到三维模型的信息。分界线传达有意义的形状信息非常紧凑，可以用于基于相似性的检索。

2. 光极值线

光极值线 (Photic extremum lines) 和其他的大部分关于模型的线条不一样。大部分线条都是基于三维模型本身的特征，而光极值线则利用人类的视觉观察和感知、亮度的突然变化来表达三维模型的线条。光极值线等价于是采用漫反射光照明模型绘制网格模型后，再对所得光照图采用边缘检测算子得到的图像边缘线。但是光极值线是三维物体上定义的，而不是二位图像上定义的，也是直接在三维模型上计算得出。这样生成的线绘制出的结果更加光滑和连续。光极值线是在 3D 模型上一系列在梯度方向光照变化达到局部最大值的点。光极值线可以用于三维模型插图的线条，和其他仅仅定义了几何形状和视点位置的线条，光极值线以描绘出光照在三维模型表面上的显著变化。计算光极值线可以用一个或者多个光源且光源的位置随机，最后都可以获得理想的结果。

3. 拉普拉斯线条

拉普拉斯线 (Laplacian Lines) 和二维图像的 Laplacian - of - Gaussian (LoG) 的图像边缘检测处理类似。可以快速实时描述模型的特征线条形状，适合比较大的模型。由于拉普拉斯光照可以被分解，所以拉普拉斯法向量可以被预计算。与传统的线条抽取技术相比，拉普拉斯实时线条抽取变得既效率又简单。依靠 Laplacian 算子的优势，Laplacian lines 可以在非常复杂的模型上使用。Laplacian lines 是一种有效的视点依赖特征线条，可以用来描绘大型的三维模型来发展交互图形应用。

参考文献

- [1] Sederberg T W, Parry S R. Free - form deformation of solid geometric models [C] //ACM Siggraph Computer Graphics. ACM, 1986, 20 (4): 151 - 160.
- [2] Ju T, Schaefer S, Warren J. Mean value coordinates for closed triangular meshes [C] //ACM Transactions on Graphics (TOG). ACM, 2005, 24 (3): 561 - 566.
- [3] Lipman Y, Levin D, Cohen - Or D. Green coordinates [C] //ACM Transactions on Graphics (TOG). ACM, 2008, 27 (3): 78.
- [4] Joshi P, Meyer M, DeRose T, et al. Harmonic coordinates for character articulation [C] //ACM Transactions on Graphics (TOG). ACM, 2007, 26 (3): 71.
- [5] Lipman Y, Kopf J, Cohen - Or D, et al. GPU - assisted positive mean value coordinates for mesh deformations [C] //Symposium on Geometry Processing. 2007, 257: 117 - 123.
- [6] Li X Y, Ju T, Hu S M. Cubic Mean Value Coordinates [J]. ACM Transactions on Graphics (TOG), 2013, 32 (4): 126.
- [7] Floater M S. Mean value coordinates [J]. Computer aided geometric design, 2003, 20 (1): 19 - 27.
- [8] Hormann K, Floater M S. Mean value coordinates for arbitrary planar polygons [J]. ACM Transactions on Graphics (TOG), 2006, 25 (4): 1424 - 1441.
- [9] Joshi P, Meyer M, DeRose T, et al. Harmonic coordinates for character articulation [C] //ACM Transactions on Graphics (TOG). ACM, 2007, 26 (3): 71.
- [10] Zhang H, Van Kaick O, Dyer R. Spectral mesh processing [C] //Computer graphics forum. Blackwell Publishing Ltd, 2010, 29 (6): 1865 - 1894.
- [11] Vallet B, Lévy B. Spectral geometry processing with manifold harmonics [C] //Computer Graphics Forum. Blackwell Publishing Ltd, 2008, 27 (2): 251 - 260.
- [12] Lévy B. Laplace - beltrami eigenfunctions towards an algorithm that " understands " geometry [C] //Shape Modeling and Applications, 2006. SMI 2006. IEEE International Conference on. IEEE, 2006: 13 - 13.
- [13] Rustamov R M. Laplace - Beltrami eigenfunctions for deformation invariant shape representation [C] //Proceedings of the fifth Eurographics symposium on Geometry processing. Eurographics Association, 2007: 225 - 233.
- [14] Liu R, Zhang H. Mesh segmentation via spectral embedding and contour analysis [C] //Computer Graphics Forum. Blackwell Publishing Ltd, 2007, 26 (3): 385 - 394.
- [15] De Goes F, Goldenstein S, Velho L. A hierarchical segmentation of articulated bodies [C] //Computer graphics forum. Blackwell Publishing Ltd, 2008, 27 (5): 1349 - 1356.
- [16] Surazhsky V, Surazhsky T, Kirsanov D, et al. Fast exact and approximate geodesics on meshes [C] //ACM Transactions on Graphics (TOG). ACM, 2005, 24 (3): 553 - 560.
- [17] Vallet B, Lévy B. Spectral geometry processing with manifold harmonics [C] //Computer Graphics Forum. Blackwell Publishing Ltd, 2008, 27 (2): 251 - 260.
- [18] Sorkine O, Alexa M. As - rigid - as - possible surface modeling [C] //Symposium on Geometry processing. 2007.

- [19] Sorkine O. Laplacian mesh processing [D]. Tel Aviv University, 2006.
- [20] Nealen A, Igarashi T, Sorkine O, et al. Laplacian mesh optimization [C] //Proceedings of the 4th international conference on Computer graphics and interactive techniques in Australasia and Southeast Asia. ACM, 2006: 381 – 389.
- [21] Lipman, Sorkine, Cohen – Or, Levin, Rössl and Seidel, “Differential Coordinates for Interactive Mesh Editing “, SMI 2004
- [22] Yu, Zhou, Xu, Shi, Bao, Guo and Shum, “Mesh Editing With Poisson – Based Gradient Field Manipulation “, SIGGRAPH 2004
- [23] Sorkine O, Cohen – Or D. Least – squares meshes [C] //Shape Modeling Applications, 2004. Proceedings. IEEE, 2004: 191 – 199.
- [24] Yu Y, Zhou K, Xu D, et al. Mesh editing with poisson – based gradient field manipulation [C] //ACM Transactions on Graphics (TOG). ACM, 2004, 23 (3): 644 – 651.
- [25] Sorkine O, Cohen – Or D, Irony D, et al. Geometry – aware bases for shape approximation [J]. Visualization and Computer Graphics, IEEE Transactions on, 2005, 11 (2): 171 – 180.
- [26] Au O K C, Tai C L, Chu H K, et al. Skeleton extraction by mesh contraction [C] //ACM Transactions on Graphics (TOG). ACM, 2008, 27 (3): 44.
- [27] Desbrun M, Meyer M, Schröder P, et al. Implicit fairing of irregular meshes using diffusion and curvature flow [C] //Proceedings of the 26th annual conference on Computer graphics and interactive techniques. ACM Press/Addison – Wesley Publishing Co. , 1999: 317 – 324.
- [28] Taubin G. A signal processing approach to fair surface design [C] //Proceedings of the 22nd annual conference on Computer graphics and interactive techniques. ACM, 1995: 351 – 358.
- [29] Kazhdan M, Solomon J, Ben – Chen M. Can Mean – Curvature Flow be Modified to be Non – singular? [C] //Computer Graphics Forum. Blackwell Publishing Ltd, 2012, 31 (5): 1745 – 1754.
- [30] Sorkine O, Cohen – Or D, Toledo S. High – Pass Quantization for Mesh Encoding [C] //Symposium on Geometry Processing. 2003: 42 – 51.
- [31] Adelson T J F D E, Judd T, Durand F. Apparent Ridges for Line Drawing [J]. proceedings of ACM Transactions on Graphics, SIGGRAPH, 2007.
- [32] Xie X, He Y, Tian F, et al. An effective illustrative visualization framework based on photic extremum lines (PELs) [J]. Visualization and Computer Graphics, IEEE Transactions on, 2007, 13 (6): 1328 – 1335.
- [33] Kolomenkin M, Shimshoni I, Tal A. Demarcating curves for shape illustration [J]. ACM Transactions on Graphics (TOG), 2008, 27 (5): 157.
- [34] DeCarlo D. Depicting 3D shape using lines [C] //IS&T/SPIE Electronic Imaging. International Society for Optics and Photonics, 2012: 829116 – 829116 – 16.
- [35] Rusinkiewicz S. Estimating curvatures and their derivatives on triangle meshes [C] //3D Data Processing, Visualization and Transmission, 2004. 3DPVT 2004. Proceedings. 2nd International Symposium on. IEEE, 2004: 486 – 493.
- [36] DeCarlo D, Rusinkiewicz S. Highlight lines for conveying shape [C] //Proceedings of the 5th international symposium on Non – photorealistic animation and rendering. ACM, 2007: 63 – 70.
- [37] Cole F, Sanik K, DeCarlo D, et al. How well do line drawings depict shape? [C] //ACM Transactions on Graphics (TOG). ACM, 2009, 28 (3): 28.
- [38] Zhang L, He Y, Xie X, et al. Laplacian lines for real – time shape illustration [C] //Proceedings of the 2009 symposium on Interactive 3D graphics and games. ACM, 2009: 129 – 136.
- [39] DeCarlo D, Finkelstein A, Rusinkiewicz S, et al. Suggestive contours for conveying shape [C] //ACM

- Transactions on Graphics (TOG). ACM, 2003, 22 (3): 848-855.
- [40] Cole F, Golovinskiy A, Limpaecher A, et al. Where do people draw lines? [C] //ACM Transactions on Graphics (TOG). ACM, 2008, 27 (3): 88.
- [41] Sousa M C, Prusinkiewicz P. A few good lines Suggestive drawing of 3d models [C] //Computer Graphics Forum. Blackwell Publishing, Inc, 2003, 22 (3): 381-390.
- [42] Yoshizawa S, Belyaev A, Seidel H P. Fast and robust detection of crest lines on meshes [C] //Proceedings of the 2005 ACM symposium on Solid and physical modeling. ACM, 2005: 227-232.
- [43] Jardim E, de Figueiredo L H. A Hybrid Method for Computing Apparent Ridges [C] //SIBGRAPI. 2010: 118-125.
- [44] Rusinkiewicz S, Cole F, DeCarlo D, et al. Line drawings from 3D models [C] //ACM SIGGRAPH 2008 classes. ACM, 20

► 作者邮箱: graphicsresearch@qq.com

VR 三维技术系列

● VR三维技术系列图书简介

着重底层核心技术的讲解, 涵盖三维图形学算法的三大方面, 即建模、动画和渲染。读者通过学习本系列专业基础图书和现有成熟计算机基础编程教材, 以及三维软件使用的图书, 知识体系可以完整地覆盖数字媒体技术专业的所有课程。

书中代码采用C#编程语言, 但是本套系列图书里讲述的原理和算法不仅限于C#语言。读者可以通过示例中的代码, 采用自己熟悉的编程语言进行编程。本套系列图书包含了很多计算机图形学会议Siggraph论文里最新的、核心的、关键突破和进展的图形学算法讲解、实现与分析。

● 读者对象

虚拟现实从业人员、数字媒体专业师生、三维游戏工程师、计算机专业师生、软件工程专业师生、影视特效工程师等。



电子信息出版分社微博
<http://weibo.com/etpublish>



策划编辑: 张 迎
责任编辑: 张 迎
封面设计: 徐海燕



官方微信平台

ISBN 978-7-121-31678-4



9 787121 316784 >

定价: 99.00 元